

APPENDIX A

CERTIFICATE OF MAILING BY EXPRESS MAIL

Express Mail Label No. EL 270 011 650 US

I hereby certify under 37 CFR §1.10 that this correspondence is being deposited with the United States Postal Service as Express Mail Post Office to Addressee with sufficient postage on the date indicated below and is addressed to the Commissioner for Patents, Washington, D.C. 20231.

Date of Deposit May 2, 2001

Signature

Samantha Bell

Typed or Printed Name of Person Signing Certificate
Samantha Bell



Overview	7-2
States	7-7
Transitions	7-14
Connective Junctions	7-28
History Junctions	7-35
Action Language	7-37

Overview

What Is Meant by Notation?

A notation defines a set of objects and the rules that govern the relationships between those objects. Stateflow notation provides a common language to communicate the design information conveyed by a Stateflow diagram.

Stateflow notation consists of:

- A set of graphical objects
- A set of nongraphical text-based objects
- Defined relationships between those objects
- Action language

Motivation Behind the Notation

Chapter 3, “Creating Charts,” and Chapter 4, “Defining Events and Data,” discuss how to use the product to create the various objects. Knowing how to create the objects is the first step to designing and implementing a Stateflow diagram. The next step is understanding and using the notation to create a well-designed and efficient Stateflow diagram.

This chapter focuses on the notation: the supported relationships amongst the graphical objects and the action language that dictates the actions that can be associated with states and transitions. The Stateflow notation supports many different ways of representing desired system behavior. The representation you choose directly affects the efficiency of the generated code.

How the Notation Checked Is Checked

The parser evaluates the graphical and nongraphical objects in each Stateflow machine against the supported Stateflow notation and the action language syntax. Errors are displayed in informational pop-up windows. See “Parsing” on page 9-20 for more information.

Some aspects of the notation are verified at runtime. Using the Debugger you can detect runtime errors such as:





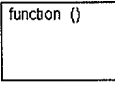



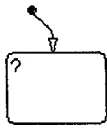



- State inconsistencies
- Conflicting transitions

- Data range violations
- Cyclic behavior

You can modify the notation to resolve runtime errors. See Chapter 10, “Debugging,” for more information on debugging runtime errors.

Graphical Objects

These are the graphical objects in the notation that are on the toolbar.

Name	Notation	Toolbar Icon
State		
Box		
Graphical Function		
History junction		
Default transition		
Connective junction		

A transition is a curved line with an arrowhead that links one graphical object to another. Either end of a transition can be attached to a source and a destination object. The *source* is where the transition begins and the *destination* is where the transition ends.

Event and data objects do not have graphical representations. These objects are defined using the Stateflow Explorer. See Chapter 4, “Defining Events and Data.”

The Data Dictionary

The data dictionary is a database containing all the information about the graphical and nongraphical objects. Data dictionary entries for graphical objects are created automatically as the objects are added and labeled. You explicitly define nongraphical objects in the data dictionary by using the Explorer. The parser evaluates entries and relationships between entries in the data dictionary to verify the notation is correct.

How Hierarchy Is Represented

The notation supports the representation of object hierarchy in Stateflow diagrams. Some of the objects are graphical while others are nongraphical.

An example of a graphical hierarchy is the ability to draw one state within the boundaries of another state. Such a representation indicates that the inner state is a substate or child of the outer state or superstate. The outer state is the parent of the inner state. In the simple case of a Stateflow diagram with a single state, the Stateflow diagram is that state's parent. Transitions are another example of graphical hierarchy. A transition's hierarchy is represented by determining its parent, source, and destination. In a Stateflow diagram you can see a transition's parent, source, and destination.

Data and event object are nongraphical and their hierarchy is represented differently (using the Explorer) from the graphical object hierarchy (using the graphics editor).

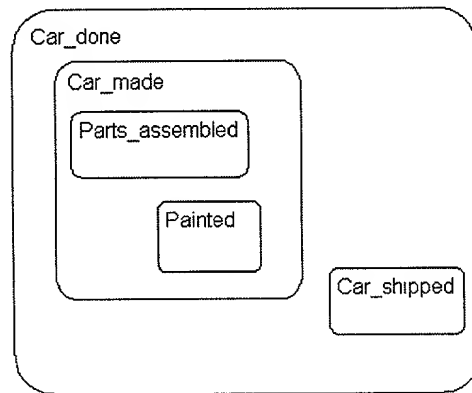
All of the objects in the notation support the representation of hierarchy.

See Chapter 4, “Defining Events and Data,” and Chapter 5, “Defining Stateflow Interfaces,” for information and examples of representing data and event objects.

For more information on how the hierarchy representations are interpreted, see Chapter 8, “Semantics.”

Example: Representing State Hierarchy

This is an example of how state hierarchy is represented.



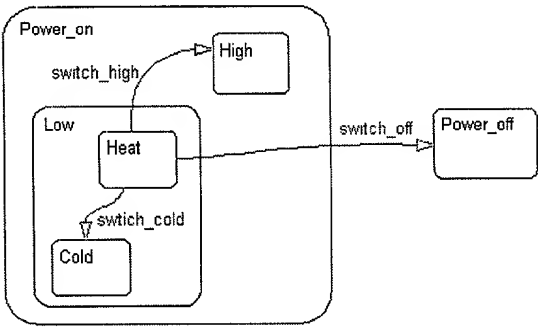
The Stateflow diagram is the parent of Car_done. Car_done is the parent state of the Car_made and Car_shipped states. Car_made is also a parent to the Parts_assembled and Car_painted states. Parts_assembled and Car_painted are children of the Car_made state.

The machine is the root of the Stateflow hierarchy. The Stateflow diagram is represented by the / character. Each level in the hierarchy of states is separated by the . character. The full hierarchy representation of the state names in this example is:

- /Car_done
- /Car_done.Car_made
- /Car_done.Car_shipped
- /Car_done.Car_made.Parts_assembled
- /Car_done.Car_made.Painted

Example: Representing Transition Hierarchy

This is an example of how transition hierarchy is represented.



A transition’s hierarchy is described in terms of the transition’s parent, source, and destination. The parent is the lowest level that the transition (source and destination) is contained within. The machine is the root of the hierarchy. The Stateflow diagram is represented by the / character. Each level in the hierarchy of states is separated by the . (period) character. The three transitions in the example are represented in the following table.

Transition Label	Parent	Source	Destination
switch_off	/	/Power_on.Low.Heat	/Power_off
switch_high	/Power_on	/Power_on.Low.Heat	/Power_on.High
switch_cold	/Power_on.Low	/Power_on.Low.Heat	/Power_on.Low.Cold

Example: Representing Event Hierarchy

Event hierarchy is defined by specifying the parent of an event when you create it. Events are nongraphical and are created using either the graphics editor **Add** menu or the Explorer. Using hierarchy you can optimize event processing through directed event broadcasting. Directed event broadcasting is the ability to qualify who can send and receive event broadcasts.

See “Defining Events” on page 4-2 for more information.

See “Action Language” on page 7-37 for more information on the notation for directed event broadcasting.


States

Overview

A *state* describes a mode of a reactive system. States in a Stateflow diagram represent these modes. The activity or inactivity of the states dynamically changes based on events and conditions.

Every state has hierarchy. In a Stateflow diagram consisting of a single state, that state's parent is the Stateflow diagram itself. A state also has history that applies to its level of hierarchy in the Stateflow diagram. States can have actions that are executed in a sequence based upon action type. The action types are: entry, during, exit, or on *event_name* actions.

This table shows the button icon and a short description of a state.

Name	Button Icon	Description
State		Use a state to depict a mode of the system.

Superstate

A state is a superstate if it contains other states, called substates.

Substate

A state is a substate if it exists in another state.

State Decomposition

A state has a *decomposition* when it consists of one or more substates. A Stateflow diagram that contains at least one state also has decomposition. Representing hierarchy necessitates some rules around how states can be grouped in the hierarchy. A superstate has either parallel (AND) or exclusive (OR) decomposition. When looking at any one point in the hierarchy, all substates of a superstate must be of the same type.

Parallel (AND) State Decomposition

Parallel (AND) state decomposition is indicated when states have dashed borders. This representation is appropriate if all states at that same level in the hierarchy are active at the same time. The activity within parallel states is essentially independent. The children of parallel (AND) decomposition parents are AND states.

Exclusive (OR) State Decomposition

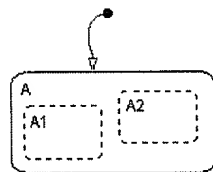
Exclusive (OR) state decomposition is represented by states with solid borders. Exclusive (OR) decomposition is used to describe system modes that are mutually exclusive. When a state has exclusive (OR) decomposition, only one substate can be active at a time. The children of exclusive (OR) decomposition parents are OR states.

Active and Inactive States

States have the Boolean characteristic of being active or inactive. The occurrence of events drives the execution of the Stateflow diagram. At any point in the execution of a Stateflow diagram, there will be some combination of active and inactive states. These are some possible combinations:

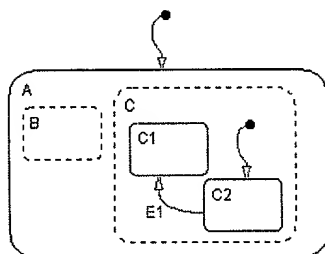
- Multiple active states with parallel (AND) decomposition

In this example, when state A is active, A1 and A2 are active.



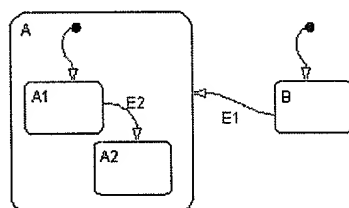
- An active state with parallel (AND) decomposition and an active state with exclusive (OR) decomposition

In this example, state B, state C, and C.C2 or state B, state C, and C.C1 are active at the same time.



- One active state with exclusive (OR) decomposition

In this example, state B or state A.A1 or state A.A2 is active at any one time.



When a given state is active, all of its ancestor states are also active. See “Semantics of Active and Inactive States” on page 8-5 for more information.

Combination States

When a Stateflow diagram has states with parallel (AND) decomposition, multiple states can be active simultaneously. A combination state is a notational representation of those multiple states. For example, a Stateflow diagram could have two active states with parallel (AND) decomposition, A.B and X.Y. Using combination state notation, the activity of the Stateflow diagram is denoted by (A.B, X.Y).

A state is characterized by its label. The label consists of the name of the state optionally followed by a / character and additional keywords defined below. The label appears on the top left-hand corner of the state rectangle.

Labeling a State

The ? character is the default state label. State labels have this general format:

```
name/  
entry:  
during:  
exit:  
on event_name:
```

The keywords entry (shorthand en), during (shorthand du), exit (shorthand ex), and on identify actions associated with the state. You can specify multiple actions by separating them by any of these:

- Carriage return
- Semicolon
- Comma

Specify multiple on *event_name* actions for different events by adding multiple on *event_name* lines specifying unique values for *event_name*.

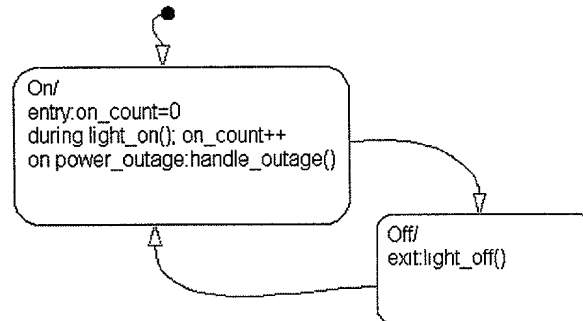
Each keyword is optional and positionally independent. You can specify none, some, or all of them. The colon after each keyword is required. The slash following the state name is optional as long as it is followed by a carriage return.

If you enter the name and slash followed directly by an action or actions (without the entry keyword), the action(s) is interpreted as entry action(s). This shorthand is useful if you are only specifying entry actions.

See “What Is an Action Language?” on page 7-37 for more information on the action language.

Example: Labeling a State

This example shows the state labeling formats and explains the components of the label.



Name. The *name* of the state forms the first part of the state label. Valid state names consist of alphanumeric characters and can include the `_` character, e.g., `Transmission` or `Green_on`.

The use of hierarchy provides some flexibility in the naming of states. The name that you enter as part of the label must be unique when preceded by the hierarchy of its ancestor states. The name as stored in the data dictionary consists of the text you entered as the label on the state, preceded by the hierarchy of its ancestor states separated by periods. States can have the same name appear on the graphical representation of the state, as long as the full names within the data dictionary are unique. The parser indicates an error if a state does not have a unique name entry in the data dictionary for that Stateflow diagram.

See “Example: Unique State Names” on page 7-12 for an example of uniquely named states.

In this example, the state names are `On` and `Off`.

Entry Action. In the example, state `On` has entry action `on_count=0`. The value of `on_count` is reset to 0 whenever state `On`’s entry action is executed.

See “Semantics of State Actions” on page 8-7 for information on how and when entry actions are executed.

During Action. In the example, state On has two during actions `light_on()` and `on_count++`. These actions are executed whenever state On's during action is executed.

See "Semantics of State Actions" on page 8-7 for information on how and when during actions are executed.

Exit Action. In the example, state Off has exit action `light_off()`. This action is executed whenever state Off's exit action is executed.

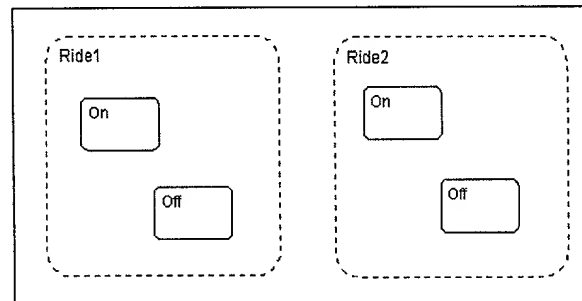
See "Semantics of State Actions" on page 8-7 for information on how and when exit actions are taken.

On Event_name Action. In the example, state Off has the on *event_name*, `power_outage`. When the event `power_outage` occurs, the action `handle_outage()` is executed.

See "Semantics of State Actions" on page 8-7 for information on how and when on *event_name* actions are taken.

Example: Unique State Names

This example shows how hierarchy supports unique naming of states.



Each of these states has a unique name because of the hierarchy of the Stateflow diagram. Although the name portion of the label on the state itself is not unique, when the hierarchy is prepended to the name in the data dictionary, the result is unique. The full names for the states as seen in the data dictionary are:

- Ride1.On
- Ride1.Off

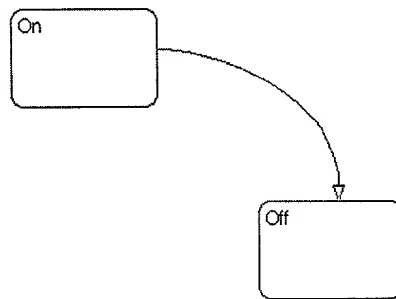
- Ride2.On
- Ride2.Off

Although the names On and Off are duplicated, the full names are unique because of the hierarchy of the Stateflow diagram. The example intentionally contains only states for simplification purposes.

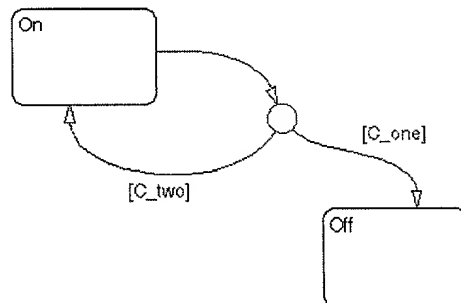
Stateflow diagram showing the hierarchy of states. The diagram is a tree structure with the root state 'Ride2' branching into 'On' and 'Off'. 'On' further branches into 'Ride2.On' and 'Ride2.Off'. 'Off' branches into 'Ride2.Off' and 'Ride2.Off'.

Transitions

In most cases, a *transition* represents the passage of the system from a source object to a destination object. There are transitions between states. There are also transitions between junctions and states. A transition is represented by a line segment ending with an arrow drawn from a source object to the destination object. This is an example of a transition from a source state, On, to a destination state, Off.



Junctions divide a transition into segments. Each segment is evaluated in the process of determining the validity of the transition from a source to a destination. This is an example of a transition with segments.



A default transition is one special type of transition that has no source object.

Labeling a Transition

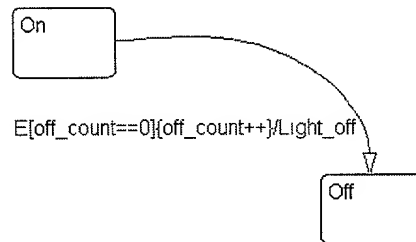
A transition is characterized by its *label*. The label can consist of an event, a condition, a condition action, and/or a transition action. The ? character is the default transition label. Transition labels have this general format.

```
event [condition]{condition_action}/transition_action
```

Replace, as appropriate, your names for event, condition, condition action, and transition action. Each part of the label is optional.

Example: Transition Label

This example shows the format of a transition label.



Event. The specified *event* is what causes the transition to be taken, provided the condition, if specified, is true. Specifying an event is optional. Absence of an event indicates that the transition is taken upon the occurrence of any event. Multiple events are specified using the OR logical operator (|).

In this example, the broadcast of event E, triggers the transition from On to Off, provided the condition, [off_count==0], is true.

Condition. A *condition* is a Boolean expression to specify that a transition occurs given that the specified expression is true. Enclose the condition in square brackets. See “Conditions” on page 7-59 for information on the condition notation.

In this example, the condition [off_count==0] must evaluate as true for the condition action to be executed and for transition from the source to the destination to be valid.

Condition Action. The *condition action* is executed as soon as the condition, if specified, is evaluated as true and before the transition destination has been determined to be valid.

If the transition consists of multiple segments, the condition action is executed as soon as the condition, if specified, is evaluated as true and before the entire transition is determined as valid. Enclose the condition action in curly brackets. See “Action Language” on page 7-37 for more information on the action language.

If no condition is specified, the implied condition is always evaluated as true.

In this example, if the condition `[off_count==0]` is true, the condition action, `off_count++` is immediately executed.

Transition Action. The *transition action* is executed after the transition destination has been determined to be valid provided the condition, if specified, is true. If the transition consists of multiple segments, the transition action is only executed when the entire transition path to the final destination is determined as valid. Precede the transition action with a backslash. See “Action Language” on page 7-37 for more information on the action language.

In this example, if the condition `[off_count==0]` is true, and the destination state `Off` is valid, the transition action `Light_off` is executed.

Valid Transitions

In most cases, a transition is valid when the source state of the transition is active and the transition label is valid. Default transitions are slightly different because there is no source state. Validity of a default transition to a substate is evaluated when there is a transition to its superstate assuming the superstate is active. This labeling criterion applies to both default transitions and general case transitions. These are possible combinations of valid transition labels.

Transition Label	Is Valid If:
Event only	That event occurs
Event and condition	That event occurs and the condition is true
Condition only	Any event occurs and the condition is true

Transition Label	Is Valid If:
Action only	Any event occurs
Not specified	Any event occurs

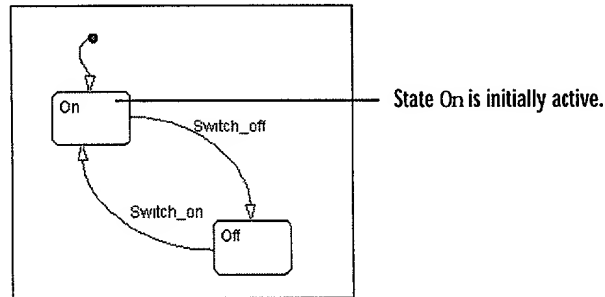
Types of Transitions

The notation supports these transition types:

- Transitions to and from exclusive (OR) states
See “Example: Transitions to and from Exclusive (OR) States” on page 7-18 for an example of this type of transition.
- Transitions to and from junctions
See “Example: Transitions to and from Junctions” on page 7-18 for an example of this type of transition.
- Transitions to and from exclusive (OR) superstates
See “Example: Transitions to and from Exclusive OR Superstates” on page 7-19 for an example of this type of transition.
- Transitions from no source to an exclusive (OR) state (default transitions)
See “Default Transitions” on page 7-21 for examples of this type of transition.
- Inner state transitions
See “What Is an Inner Transition?” on page 7-24 for examples of this type of transition.
- Self loop transitions
See “What Is a Self Loop Transition?” on page 7-27 for examples of this type of transition.

Example: Transitions to and from Exclusive (OR) States

This example shows simple transitions to and from exclusive (OR) states.

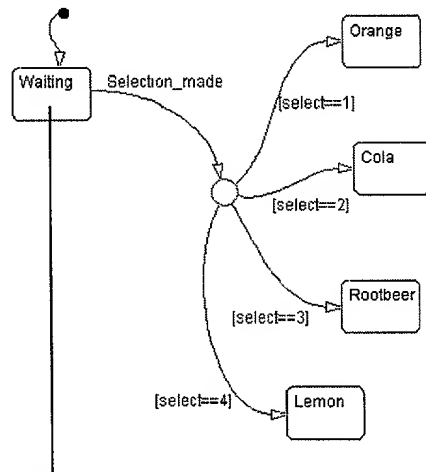


The transition On→Off is valid when state On is active and the event Switch_off occurs. The transition Off→On is valid when state Off is active and event Switch_on occurs.

See “Transitions to and from Exclusive (OR) States” on page 8-8 for more information on the semantics of this notation.

Example: Transitions to and from Junctions

This example shows transitions to and from a connective junction.



State Waiting is initially active.

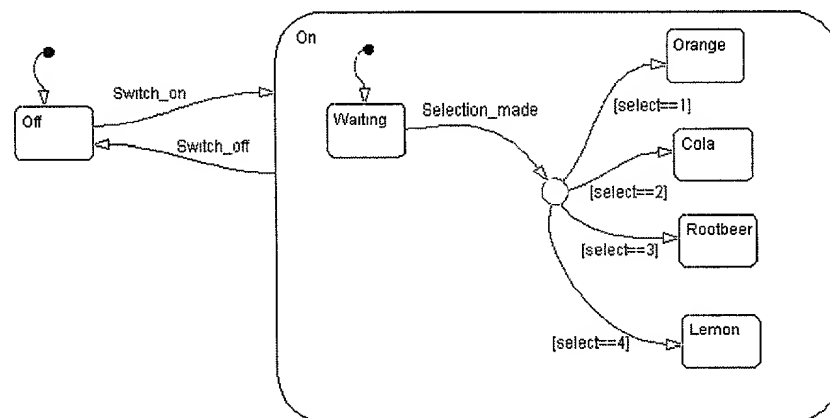
This is a Stateflow diagram of a soda machine. The Stateflow diagram is called when the external event Selection_made occurs. The Stateflow diagram

awakens with the Waiting state active. The Waiting state is a common source state. When the event Selection_made occurs, the Stateflow diagram transitions from the Waiting state to one of the other states based on the value of the variable select. One transition is drawn from the Waiting state to the connective junction. Four additional transitions are drawn from the connective junction to the four possible destination states.

See “Example: Transitions from a Common Source to Multiple Destinations” on page 8-36 for more information on the semantics of this notation.

Example: Transitions to and from Exclusive OR Superstates

This example shows transitions to and from an exclusive (OR) superstate and the use of a default transition.



This is an expansion of the soda machine Stateflow diagram that includes the initial example of the On and Off exclusive (OR) states. On is now a superstate containing the Waiting and soda choices states. The transition Off→On is valid when state Off is active and event Switch_on occurs. Now that On is a superstate, this is an explicit transition to the On superstate.

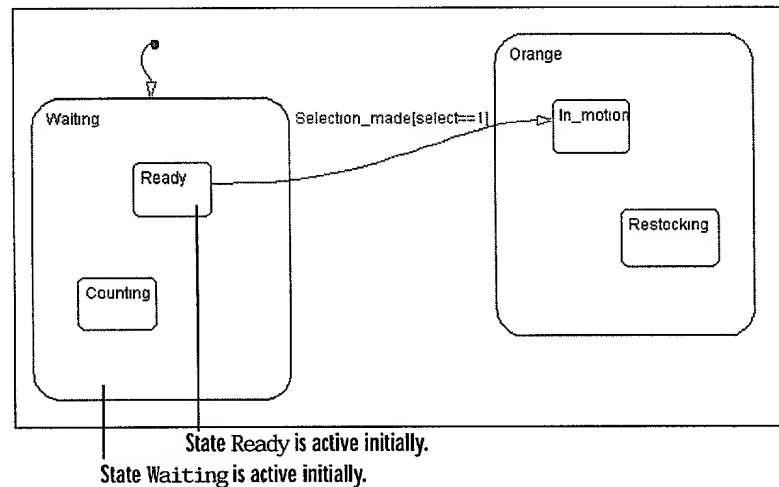
To be a valid transition to a superstate, the destination substate must be implicitly defined. By defining that the Waiting substate has a default transition, the destination substate is implicitly defined. This notation defines that the resultant transition is Off→On.Waiting.

The transition On→Off is valid when state On is active and event Switch_off occurs. When the Switch_off event occurs, no matter which of the substates of On is active, we want to transition to the Off state. This top-down approach supports the ability to simplify the Stateflow diagram by looking at the transitions out of the superstate without considering all the details of states and transitions within the superstate.

See “Default Transitions” on page 8-18 for more information on the semantics of this notation.

Example: Transitions to and from Substates

This example shows transitions to and from exclusive (OR) substates.



Two of the substates of the On superstate are further defined to be superstates of their own. The Stateflow diagram shows a transition from one OR substate to another OR substate. The transition Waiting.Ready→Orange.In_motion is valid when state Waiting.Ready is active and event Selection_made occurs, providing that the select variable equals one. This transition defines an explicit exit from the Waiting.Ready state and an implicit exit from the Waiting superstate. On the destination side, this transition defines an implicit entry into the Orange superstate and an explicit entry into the Orange.In_motion substate.

See “Example: Transition from a Substate to a Substate” on page 8-11 for more information on the semantics of this notation.


Default Transitions

Default transitions are primarily used to specify which exclusive (OR) state is to be entered when there is ambiguity among two or more neighboring exclusive (OR) states. For example, default transitions specify which substate of a superstate with exclusive (OR) decomposition the system enters by default in the absence of any other information such as a history junction. Default transitions are also used to specify that a junction should be entered by default. The default transition object is a transition with a destination but no source object.

Click on the **Default transition** button in the toolbar, and click on a location in the drawing area close to the state or junction you want to be the destination for the default transition. Drag the mouse to the destination object to attach the default transition. In some cases it is useful to label default transitions.

One of the most common Stateflow programming mistakes is to create multiple exclusive (OR) states without a default transition. In the absence of the default transition, there is no indication of which state becomes active by default. Note that this error is flagged when you simulate the model using the Debugger with the **State Inconsistencies** option enabled.

This table shows the button icon and briefly describes a default transition.

Name	Button Icon	Description
Default transition		Use a default transition to indicate, when entering this level in the hierarchy, which object becomes active by default.

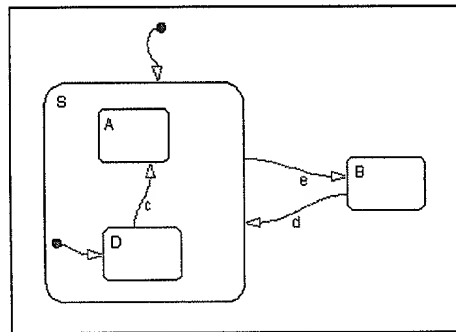
Labeling Default Transitions

In some circumstances, you may want to label default transitions. You can label default transitions as you would other transitions. For example, you may want to specify that one state or another should become active depending upon the event that has occurred. In another situation, you may want to have specific actions take place that are dependent upon the destination of the transition.

Note When labeling default transitions, take care to ensure that there will always be at least one valid default transition. Otherwise, the state machine can transition into an inconsistent state.

Example: Use of Default Transitions

This example shows a use of default transitions.



When the Stateflow diagram is first awakened, the default transition to superstate S defines that of states S and B; the transition to state S is valid. State S has two substates, A and D. Which substate does the system transfer to? It cannot transfer to both of them since A and D are not parallel (AND) states. Again, this kind of ambiguity is cleared up by defining a default transition to substate D.

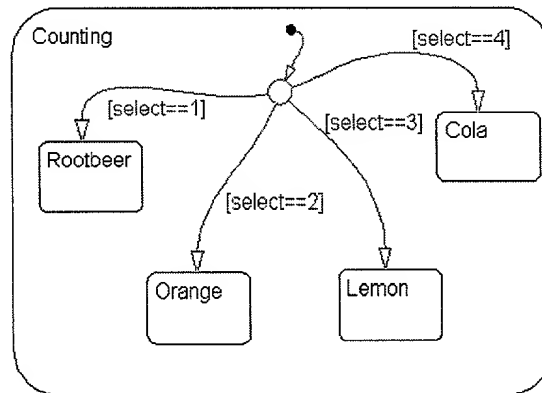
Suppose at a different execution point, the Stateflow diagram is awakened by the occurrence of event d and state B is active. The transition B→S is valid. When the system enters state S, it enters substate D because the default transition is defined.

See “Default Transitions” on page 8-18 for more information on the semantics of this notation.

The default transitions are required for the Stateflow diagram to execute. Without the default transition to state S, when the Stateflow diagram is awakened, none of the states become active. You can detect this situation at runtime by checking for state inconsistencies. See “Animation Controls” on page 10-8 for more information.

Example: Default Transition to a Junction

This example shows a default transition to a connective junction.

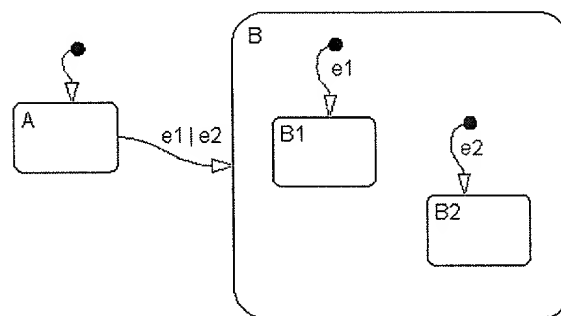


In this example, the default transition to the connective junction defines that upon entering the Counting state, the destination is determined by the condition on each transition segment.

See "Example: Default Transition to a Junction" on page 8-19 for more information on the semantics of this notation.

Example: Default Transition with a Label

This example shows a use of labeling default transitions.



If state A is initially active and either e1 or e2 occurs, the transition from state A to superstate B is valid. The substates B1 and B2 both have default transitions. The default transitions are labeled to specify the event that triggers the transition. If event e1 occurs, the transition A→B1 is valid. If event e2 occurs, the transition A→B2 is valid.

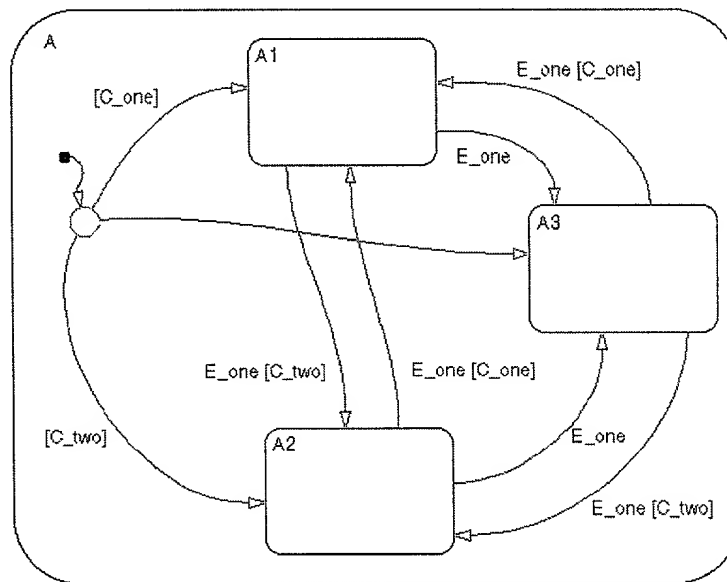
See “Example: Labeled Default Transitions” on page 8-21 for more information on the semantics of this notation.

What Is an Inner Transition?

An *inner transition* is a transition that does not exit the source state. Inner transitions are most powerful when defined for superstates with exclusive (OR) decomposition. Use of inner transitions can greatly simplify a Stateflow diagram.

Example One: Before Using an Inner Transition

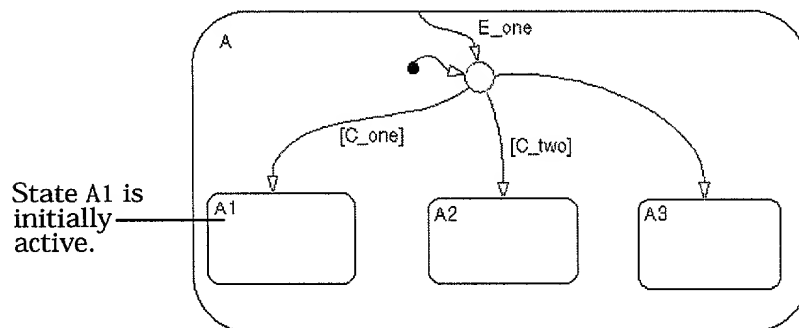
This is an example of a Stateflow diagram that could be simplified by using an inner transition.



Any event occurs and awakens the Stateflow diagram. The default transition to the connective junction is valid. The destination of the transition is determined by [C_one] and [C_two]. If [C_one] is true, the transition to A1 is true. If [C_two] is true, the transition to A2 is valid. If neither [C_one] nor [C_two] is true, the transition to A3 is valid. The transitions among A1, A2, and A3 are determined by E_one, [C_one], and [C_two].

Example One: Inner Transition to a Connective Junction

This example shows a solution to the same problem (Example One) using an inner transition to a connective junction.



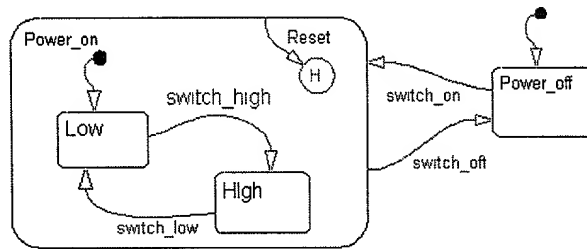
Any event occurs and awakens the Stateflow diagram. The default transition to the connective junction is valid. The destination of the transitions is determined by [C_one] and [C_two].

The Stateflow diagram is simplified by using an inner transition in place of the many transitions amongst all the states in the original example. If state A is already active, the inner transition is used to re-evaluate which of the substates of state A is to be active. When event E_one occurs, the inner transition is potentially valid. If [C_one] is true, the transition to A1 is valid. If [C_two] is true, the transition to A2 is valid. If neither [C_one] nor [C_two] is true, the transition to A3 is valid. This solution is much simpler than the previous one.

See "Example: Processing One Event with an Inner Transition to a Connective Junction" on page 8-26 for more information on the semantics of this notation.

Example: Inner Transition to a History Junction

This example shows an inner transition to a history junction.



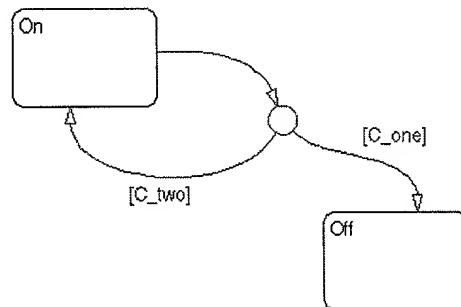
State **Power_on.High** is initially active. When event **Reset** occurs, the inner transition to the history junction is valid. Because the inner transition is valid, the currently active state, **Power_on.High**, will be exited. When the inner transition to the history junction is processed, the last active state, **Power_on.High**, becomes active (is re-entered). If **Power_on.Low** was active under the same circumstances, **Power_on.Low** would be exited and re-entered as a result. The inner transition in this example is equivalent to drawing an outer self-loop transition on both **Power_on.Low** and **Power_on.High**.

See “Example: Use of History Junctions” on page 7-35 for another example using a history junction.

See “Example: Inner Transition to a History Junction” on page 8-29 for more information on the semantics of this notation.

What Is a Self Loop Transition?

A transition segment from a state to a connective junction that has an outgoing transition segment from the connective junction back to itself is a self loop. This is an example of a self loop.



See these sections for examples of self loops:

- “Example: Connective Junction Special Case - Self Loop” on page 7-30
See “Example: Self Loop” on page 8-32 for information on the semantics of this notation.
- “Example: Connective Junction and For Loops” on page 7-31
See “Example: For Loop Construct” on page 8-33 for information on the semantics of this notation.

Connective Junctions

What Is a Connective Junction?

A connective junction is used to represent a decision point in the Stateflow diagram. The connective junction enables representation of different transition paths. Connective junctions are used to help represent:

- Variations of an if-then-else decision construct by specifying conditions on some or all of the outgoing transitions from the connective junction.
- A self loop back to the source state if none of the outgoing transitions is valid.
- Variations of a for loop construct by having a self loop transition from the connective junction back to itself.
- Transitions from a common source to multiple destinations.
- Transitions from multiple sources to a common destination.
- Transitions from a source to a destination based on common events

See “Connective Junctions” on page 8-31 for a summary of the semantics of connective junctions.

What Is Flow Diagram Notation?

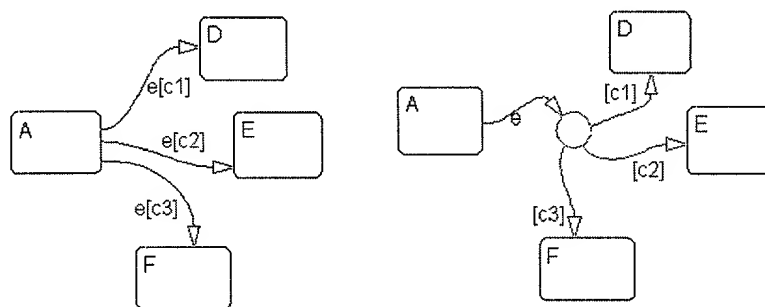
Flow diagram notation is essentially logic represented without the use of states. In some cases, using flow diagram notation is a closer representation of the system's logic and avoids the use of unnecessary states. Flow diagram notation is an effective way to represent common code structures like for loops and if-then-else constructs. The use of flow diagram notation in a Stateflow diagram can produce more efficient code optimized for memory use. Reducing the number of states optimizes memory use.

Flow diagram notation is represented through combinations of self-loops to connective junctions, transitions to and from connective junctions, and inner transitions to connective junctions. The key to representing flow diagram notation is in the labeling of the transitions (specifically the use of action language).

Flow diagram notation and state-to-state transition notation seamlessly coexist in the same Stateflow diagram.

Example: Connective Junction with All Conditions Specified

When event e occurs, state A transfers to D , E , or F depending on which of the conditions $[c1]$, $[c2]$, or $[c3]$ is met. With the alternative representation, using a connective junction, the transition from A to the connective junction occurs first, provided the event has occurred. A destination state is then determined based on which of the conditions $[c1]$, $[c2]$, or $[c3]$ is satisfied. The transition from the source state to the connective junction is labeled by the event, and those from the connective junction to the destination states by the conditions. No event is applicable in a transition from a connective junction to a destination state.

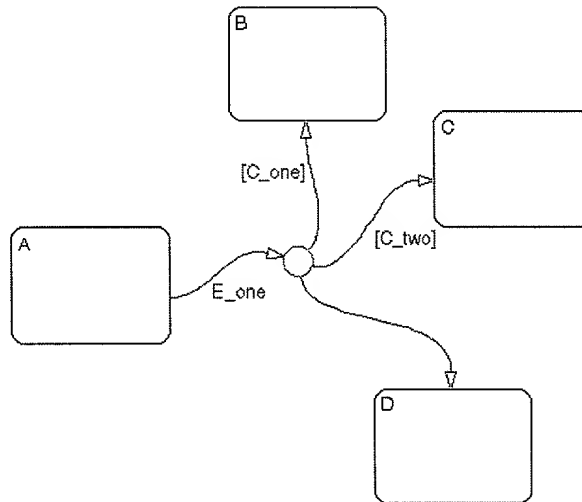


See “Example: If-Then-Else Decision Construct” on page 8-31 for information on the semantics of this notation.

Example: Connective Junction with One Unconditional Transition

The transition $A \rightarrow B$ is valid when A is active, event E_one occurs, and $[C_one]$ is true. The transition $A \rightarrow C$ is valid when A is active, event E_one occurs, and $[C_two]$ is true. Otherwise, given A is active and event E_one occurs, the

transition A→D is valid. If you do not explicitly specify condition [C_three], it is implicit that the transition condition is not [C_one] and not [C_two].



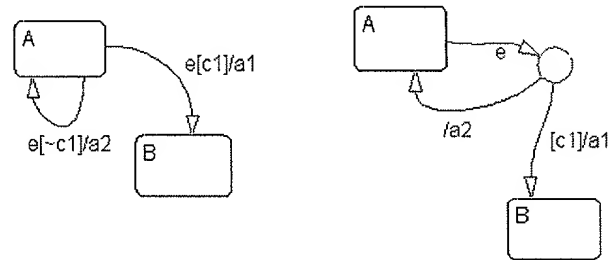
See “Example: If-Then-Else Decision Construct” on page 8-31 for information on the semantics of this notation.

Example: Connective Junction Special Case - Self Loop

In some situations, the transition event occurs, but the condition is not met. The transition cannot be taken, but an action is generated. You can represent this situation by using a connective junction or a self loop (transition from state to itself).

In state A, event e occurs. If condition [c1] is met, transition A→B is taken, generating action a1. The transition A→A is valid if event e occurs and [c1] is not true. In this self loop, the system exits and re-enters state A, and executes action a2. An alternative representation using a connective junction is shown.

The two representations are equivalent; in the one that uses a connective junction, it is not necessary to specify condition $[\sim c1]$ explicitly, as it is implied.



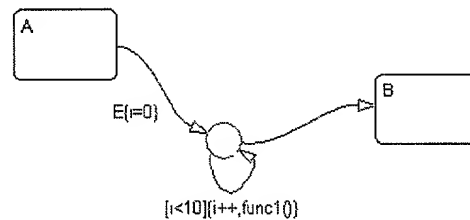
See “Example: Self Loop” on page 8-32 for information on the semantics of this notation.

Example: Connective Junction and For Loops

This example shows a combination of flow diagram notation and state transition notation. Self loops to connective junctions can be used to represent for loop constructs.

In state A, event E occurs. The transition from state A to state B is valid if the conditions along the transition path are true. The first segment of the transition does not have a condition, but does have a condition action. The condition action, $\{i=0\}$, is executed. The condition on the self loop is evaluated as true and the condition actions $\{i++; func1()\}$ execute. The condition actions execute until the condition, $[i < 10]$, is false. The condition actions on both the first segment and the self loop to the connective junction effectively execute a for loop (for i values 0 to 9 execute $func1()$). The for loop is executed outside of the context of a state. The remainder of the path is

evaluated. Since there are no conditions, the transition completes at the destination, state B.



See “Example: For Loop Construct” on page 8-33 for information on the semantics of this notation.

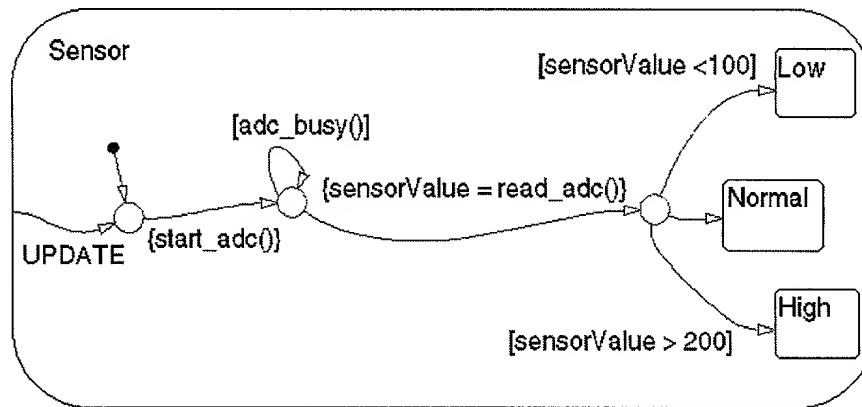
Example: Flow Diagram Notation

This example shows a real-world use of flow diagram notation and state transition notation. This Stateflow diagram models an 8-bit analog-to-digital converter (ADC).

Consider the case when state `Sensor.Low` is active and event `UPDATE` occurs. The inner transition from `Sensor` to the connective junction is valid. The next transition segment has a condition action, `{start_adc() }`, which initiates a reading from the ADC. The self-loop on the second connective junction repeatedly tests the condition `[adc_busy()]`. This condition evaluates as true once the reading settles (stabilizes) and the loop completes. This self loop is used to introduce the delay needed for the ADC reading to settle. The delay could have been represented by using another state with some sort of counter. Using flow notation in this example avoids an unnecessary use of a state and produces more efficient code.

The next transition segment condition action, `{sensorValue=read_adc() }`, puts the new value read from the ADC in the data object `sensorValue`. The final transition segment is determined by the value of `sensorValue`. If `[sensorValue <100]` is true, the state `Sensor.Low` is the destination. If

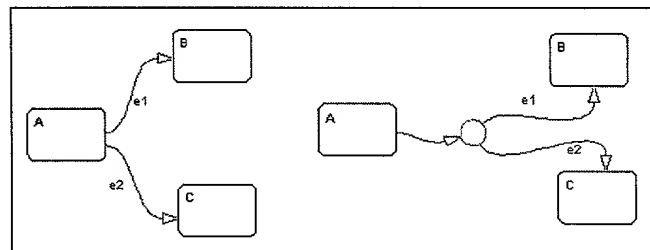
[sensorValue > 200] is true, the state Sensor.High is the destination. Otherwise, state Sensor.Normal is the destination state.



See “Example: Flow Diagram Notation” on page 8-34 for information on the semantics of this notation.

Example: Connective Junction from a Common Source to Multiple Destinations

Transitions $A \rightarrow B$ and $A \rightarrow C$ share a common source state A. An alternative representation uses one arrow from A to a connective junction, and multiple arrows labeled by events from the junction to the destination states B and C.



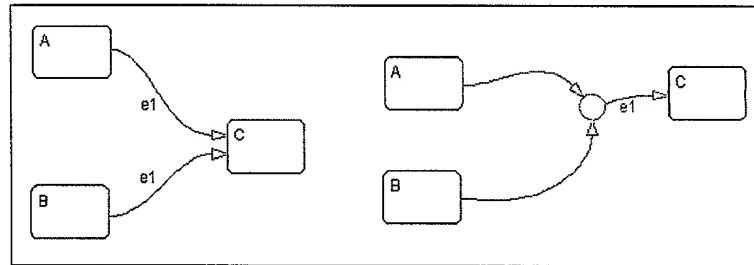
See “Example: Transitions from a Common Source to Multiple Destinations” on page 8-36 for information on the semantics of this notation.

Example: Connective Junction Common Events

Suppose, for example, that when event $e1$ occurs, the system, whether it is in state A or B, will transfer to state C. Suppose that transitions $A \rightarrow C$ and $B \rightarrow C$ are triggered by the same event $e1$, so that both destination state and trigger event are common between the transitions. There are three ways to represent this:

- By drawing transitions from A and B to C, each labeled with $e1$
- By placing A and B in one superstate S, and drawing one transition from S to C, labeled with $e1$
- By drawing transitions from A and B to a connective junction, then drawing one transition from the junction to C, labeled with $e1$

This Stateflow diagram shows the simplification using a connective junction.



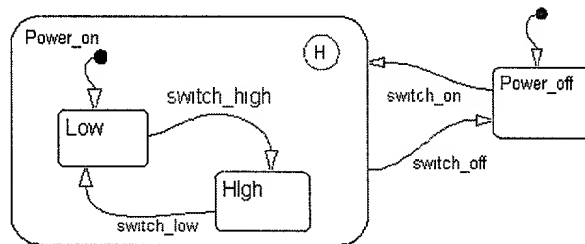
See “Example: Transitions from a Source to a Destination Based on a Common Event” on page 8-38 for information on the semantics of this notation.

History Junctions

A history junction is used to represent historical decision points in the Stateflow diagram. The decision points are based on historical data relative to state activity. Placing a history junction in a superstate indicates that historical state activity information is used to determine the next state to become active. The history junction applies only to the level of the hierarchy in which it appears.

Example: Use of History Junctions

This example shows a use of history junctions.



Superstate `Power_on` has a history junction and contains two substates. If state `Power_off` is active and event `switch_on` occurs, the system could enter either `Power_on.Low` or `Power_on.High`. The first time superstate `Power_on` is entered, substate `Power_on.Low` will be entered because it has a default transition. At some point afterwards, if state `Power_on.High` is active and event `switch_off` occurs, superstate `Power_on` is exited and state `Power_off` becomes active. Then event `switch_on` occurs. Since `Power_on.High` was the last active state, it becomes active again. After the first time `Power_on` becomes active, the choice between entering `Power_on.Low` or `Power_on.High` is determined by the history junction.

See “Example: Default Transition and a History Junction” on page 8-20 for more information on the semantics of this notation.

History Junctions and Inner Transitions

By specifying an inner transition to a history junction, you can specify that, based on a specified event and/or condition, the active state is to be exited and then immediately re-entered.

See “Example: Inner Transition to a History Junction” on page 7-26 for an example of this notation.

See “Example: Inner Transition to a History Junction” on page 8-29 for more information on the semantics of this notation.

See “Example: Inner Transition to a History Junction” on page 8-29 for more information on the semantics of this notation.

Action Language

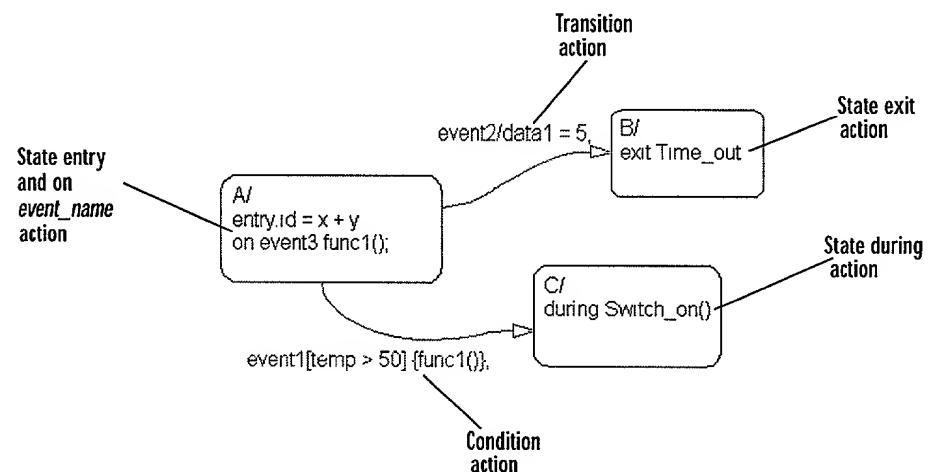
What Is an Action Language?

You sometimes want actions to take place as part of Stateflow diagram execution. The action can be executed as part of a transition from one state to another, or it can depend on the activity status of a state. Transitions can have condition actions and transition actions. States can have entry, during, exit, and, on *event_name* actions.

An action can be a function call, an event to be broadcast, a variable to be assigned a value, etc. The *action language* defines the categories of actions you can specify and their associated notations. Violations of the action language notation are flagged as errors by the parser. This section describes the action language notation rules.

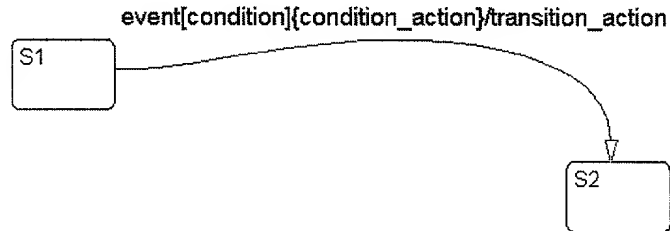
Objects with Actions

This Stateflow diagram shows examples of the possible transition and state actions.



Transition Action Notation

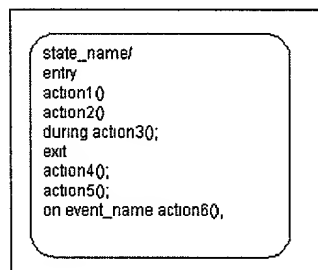
Actions can be associated with transitions via the transition's label. The general format of a transition label is shown below.



When the event occurs, the transition is evaluated. The condition action is executed as soon as the condition is evaluated as true and before the transition destination has been determined to be valid. Enclose the condition action in curly brackets. Specifying a transition action means that the action is executed when the transition is taken, provided the condition, if specified, is true.

State Action Notation

Actions can be associated with states via the state's label by defining entry, during, exit, and on *event_name* keywords. The general format of a state label is shown below.



The / (forward slash) following the state name is optional. See “Semantics of State Actions” on page 8-7 for information on the semantics of state actions. See the examples of the semantics of state actions in Chapter 8, “Semantics,” .

Keywords

These Stateflow keywords have special meaning in the notation.

Keyword	Shorthand	Meaning
<code>change(<i>data_name</i>)</code>	<code>chg(<i>data_name</i>)</code>	Generates a local event when the value of <i>data_name</i> changes.
<code>during</code>	<code>du</code>	Actions that follow are executed as part of a state's during action.
<code>entry</code>	<code>en</code>	Actions that follow are executed as part of a state's entry action.
<code>entry(<i>state_name</i>)</code>	<code>en(<i>state_name</i>)</code>	Generates a local event when the specified <i>state_name</i> is entered.
<code>exit</code>	<code>ex</code>	Actions that follow are executed as part of a state's exit action.
<code>exit(<i>state_name</i>)</code>	<code>ex(<i>state_name</i>)</code>	Generates a local event when the specified <i>state_name</i> is exited.
<code>in(<i>state_name</i>)</code>	<code>none</code>	A condition function that is evaluated as true when the <i>state_name</i> specified as the argument is active.
<code>on <i>event_name</i></code>	<code>none</code>	Actions that follow are executed when the <i>event_name</i> specified as an argument to the on keyword is broadcast.

Keyword	Shorthand	Meaning
<code>send(event_name, state_name)</code>	<code>none</code>	Send the <code>event_name</code> specified to the <code>state_name</code> specified (directed event broadcasting).
<code>matlab(evalString, arg1, arg2, ...)</code>	<code>m1()</code>	Action specifies a call using MATLAB function notation.
<code>matlab.MATLAB_workspace_data</code>	<code>m1.</code>	Action specifies a call using the <code>m1</code> name space notation.

Note Use of these keywords in any way other than their intended meaning within the rules of the notation will cause unpredictable results.

Action Language Components

See the following sections for descriptions and usage of action language components:

- “Bit Operations” on page 7-41
- “Binary Operations” on page 7-42
- “Unary Operations” on page 7-44
- “Unary Actions” on page 7-44
- “User-Written Functions” on page 7-45
- “`m1()` Functions” on page 7-47
- “MATLAB Name Space Operator” on page 7-50
- “Data and Event Arguments” on page 7-53
- “Arrays” on page 7-53
- “Pointer and Address Operators” on page 7-54
- “Hexadecimal Notation” on page 7-55
- “Typecast Operators” on page 7-55
- “Event Broadcasting” on page 7-56

- “Directed Event Broadcasting” on page 7-57
- “Conditions” on page 7-59
- “Time Symbol” on page 7-60
- “Literals” on page 7-60
- “Continuation Symbols” on page 7-61
- “Comments” on page 7-61
- “Use of the Semicolon” on page 7-61
- “Temporal Logic Operators” on page 7-61
- “Temporal Logic Events” on page 7-66

Bit Operations

You can enable C-like bit operations. See “Preserve symbol names” on page 9-14 for more information. If you have `bitops` enabled, some of the logical binary operators and unary operators are interpreted as bitwise operators. See “Binary Operations” on page 7-42 and “Unary Operations” on page 7-44 for specific interpretations.

Binary Operations

Binary operations fall into these categories.

Numerical

Example	Description
<code>a + b</code>	Addition of two operands
<code>a - b</code>	Subtraction of one operand from the other
<code>a * b</code>	Multiplication of two operands
<code>a / b</code>	Division of one operand by the other
<code>a %% b</code>	Modulus

Logical

(The default setting; bit operations are not enabled.)

Example	Description
<code>a == b</code>	Comparison of equality of two operands
<code>a & b</code>	Logical AND of two operands
<code>a && b</code>	
<code>a b</code>	Logical OR of two operands
<code>a b</code>	
<code>a ~= b</code>	Comparison of inequality of two operands
<code>a != b</code>	
<code>a > b</code>	Comparison of the first operand greater than the second operand
<code>a < b</code>	Comparison of the first operand less than the second operand

Example	Description
<code>a >= b</code>	Comparison of the first operand greater than or equal to the second operand
<code>a <= b</code>	Comparison of the first operand less than or equal to the second operand

Logical

(Bit operations are enabled.)

Example	Description
<code>a == b</code>	Comparison of equality of two operands
<code>a && b</code>	Logical AND of two operands
<code>a & b</code>	Bitwise AND of two operands
<code>a b</code>	Logical OR of two operands
<code>a b</code>	Bitwise OR of two operands
<code>a ~= b</code>	Comparison of inequality of two operands
<code>a != b</code>	
<code>a <> b</code>	
<code>a > b</code>	Comparison of the first operand greater than the second operand
<code>a < b</code>	Comparison of the first operand less than the second operand
<code>a >= b</code>	Comparison of the first operand greater than or equal to the second operand
<code>a <= b</code>	Comparison of the first operand less than or equal to the second operand
<code>a ^ b</code>	Bitwise XOR of two operands

Unary Operations

These unary operations are supported: `~`, `!`, `-`.

Example	Description
<code>~a</code>	Logical not of a Complement of a (if bitops is enabled)
<code>!a</code>	Logical not of a
<code>-a</code>	Negative of a

Unary Actions

These unary actions are supported.

Example	Description
<code>a++</code>	Increment a
<code>a--</code>	Decrement a

Assignment Operations

These assignment operations are supported.

Example	Description
<code>a = expression</code>	Simple assignment
<code>a += expression</code>	Equivalent to <code>a = a + expression</code>
<code>a -= expression</code>	Equivalent to <code>a = a - expression</code>
<code>a *= expression</code>	Equivalent to <code>a = a * expression</code>
<code>a /= expression</code>	Equivalent to <code>a = a / expression</code>

These additional assignment operations are supported when bit operations are enabled.

Example	Description
<code>a = expression</code>	Equivalent to <code>a = a expression</code> (bit operation)
<code>a &= expression</code>	Equivalent to <code>a = a & expression</code> (bit operation)
<code>a ^= expression</code>	Equivalent to <code>a = a ^ expression</code> (bit operation)

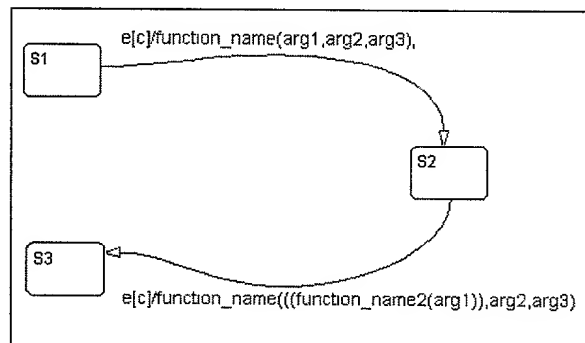
User-Written Functions

You can specify calls to user-written functions in the action language. These guidelines apply to user-written function calls:

- Define a function by its name, any arguments in parenthesis, and an optional semicolon.
- String parameters to user-written functions are passed between single quotes. For example, `func('string')`.
- An action can nest function calls.
- An action can invoke functions that return a scalar value (of type double in the case of MATLAB functions and of any type in the case of C user-written functions).

Example: Function Call Transition Action

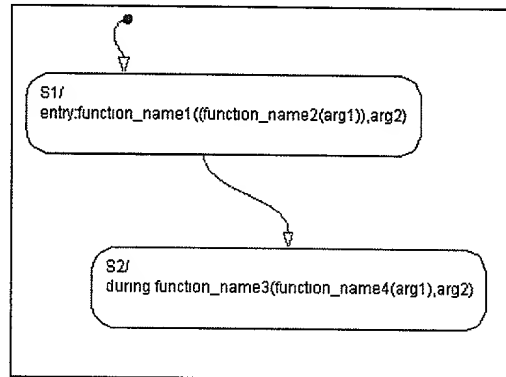
These are example formats of function calls using transition action notation.



If S1 is active, event *e* occurs, *c* is true, and the transition destination is determined, then a function call is made to `function_name` with `arg1`, `arg2`, and `arg3`. The transition action in the transition from S2 to S3 shows a function call nested within another function call.

Example: Function Call State Action

These are example formats of function calls using state action notation.



When the default transition into S1 occurs, S1 is marked active and then its entry action, a function call to `function_name1` with the specified arguments, is executed and completed. If S2 is active and an event occurs, the during action, a function call to `function_name3` with the specified arguments, executes and completes.

Passing Arguments by Reference

A Stateflow action can pass arguments to a user-written function by reference rather than by value. In particular, an action can pass a pointer to a value rather than the value itself. For example, an action could contain the following call.

```
f(&x);
```

where *f* is a custom-code C-function that expects a pointer to *x* as an argument.

If *x* is the name of a data item defined in the SF data dictionary, the following rules apply.

- Do not use pointers to pass data items input from Simulink.

If you need to pass an input item by reference, for example, an array, assign the item to a local data item and pass the local item by reference.

- If x is a Simulink output data item having a data type other than `double`, the chart property **Use strong data typing with Simulink IO** must be on (see “Specifying Chart Properties” on page 3-30).
- If the data type of x is `boolean`, the coder option **Use bitsets to store state-configuration** must be turned off (see “Use bitsets for storing state configuration” on page 9-16).
- If x is an array with its first index property set to zero (see “Array” on page 4-17), then the function must be called as follows.

```
f(&(x[0]));
```

This will pass a pointer to the first element of x to the function.

- If x is an array with its first index property set to a non-zero number (for example, 1), the function must be called in the following way.

```
f(&(x[1]));
```

This will pass a pointer to the first element of x to the function.

ml() Functions

You can specify calls to MATLAB functions that return scalars (of type `double`) in the action language.

ml() Function Format

The format of the `ml()` function is

```
ml(evalString, arg1, arg2, arg3,...);
```

where the return value is scalar (of type `double`).

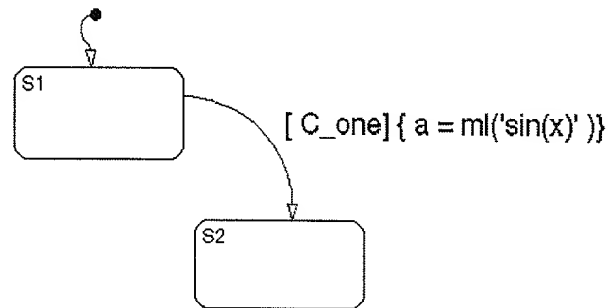
If the result returned is:

- A vector, then the first element is returned.
- A void, then an appropriate format must be used (an assignment statement cannot be used).
- A string, a structure, or a cell array, then the behavior is undefined.

`evalString` is a string that is evaluated in the MATLAB workspace with formatted substitutions of `arg1`, `arg2`, `arg3`, etc.

Example One: `ml()` Function Call

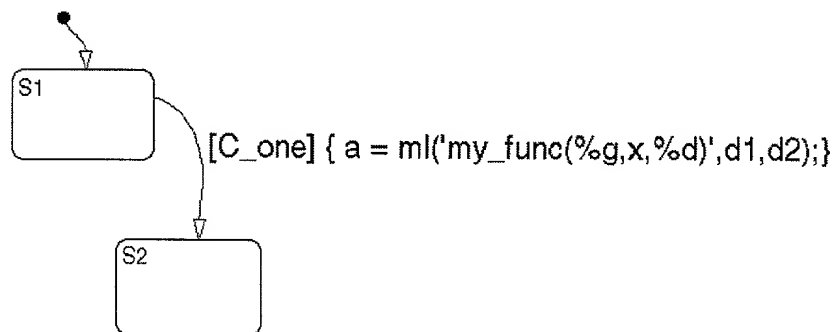
This is an example of an `ml()` function call as part of a condition action.



If `S1` is active, an event occurs, and if `[c_one]` is true, the expression `sin(x)` is evaluated in the MATLAB workspace and the return value assigned to `a`. (`x` must be a variable in the MATLAB workspace and `a` is a data object in the Stateflow diagram). The result of the evaluation must be a scalar. If `x` is not defined in the MATLAB workspace, a runtime error is generated.

Example Two: `ml()` Function Call

This is an example of a `ml()` function call that passes Stateflow data as arguments. Notice the use of format specifiers `%g` and `%d` as are used in the C language function `printf`.



These data objects are defined:

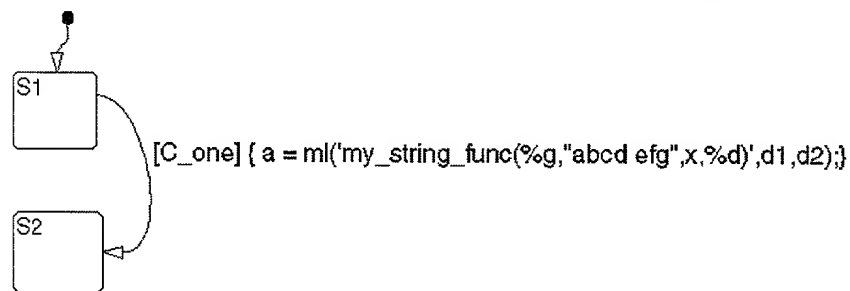
- d1 and a are **Local** data objects of type double in the Stateflow diagram
- d2 is an **Output to Simulink** data object of type integer in the Stateflow diagram
- x must be defined in the MATLAB workspace prior to the execution of the condition action where it is used; if it is not defined, a runtime error is generated.

These three values are passed as arguments to a user-written function. The %g and %d characters are format specifiers that print the current values of d1 and d2 into evalString at appropriate locations.

For example if d1 equals 3.4 and d2 equals 5, using the format specifiers these are mapped into `my_func(3.4,x,5)`. This string is then sent to MATLAB and is executed in the MATLAB workspace.

Example Three: ml() Function Call

This is an example of a `ml()` function call with string arguments.



These data objects are defined in the Stateflow diagram:

- d1 is a **Local** data object of type double
- d2 is an **Output to Simulink** data object of type integer

The user-written function `my_string_func` expects four arguments, where the second argument is a string. The %g and %d characters are format specifiers that print the current values of d1 and d2 into evalString at appropriate locations. Notice that the string is enclosed in two single quotes.

Use Guidelines

These guidelines apply to `ml()` functions:

- The first argument must be a string.
- If there are multiple arguments, ensure that the number and types of format specifiers (`%g`, `%d`, etc.) match the actual number and types of the arguments. These format specifiers are the same as those used in the C function `printf`.
- A scalar (of type double) is returned.
- `ml()` function calls can be nested.
- Calls to `ml()` functions should be avoided if you plan to build an RTW target that includes code from Stateflow Coder.

MATLAB Name Space Operator

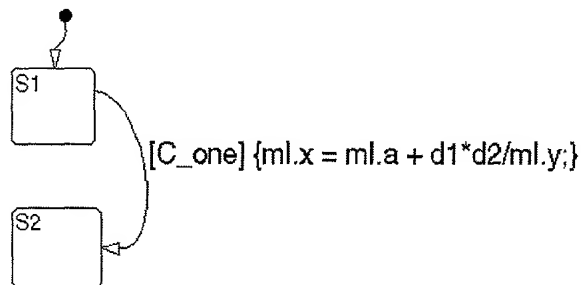
The MATLAB name space operator, `ml`, is used to get and set variables in the MATLAB workspace. The `ml` operator can also be used to access MATLAB functions that operate on scalars in a convenient format.

Use the notation, `a = ml.func_name()` ; , to call a MATLAB function that does not accept any arguments. Omission of the empty brackets causes a search for a variable of the name specified. The variable will not be found and a runtime error is encountered during simulation.

Use of the `ml` name space operator should be avoided if you plan to build a Real-Time Workshop target that includes code from Stateflow Coder.

Example: Using the ml Operator to Access MATLAB Workspace Variables

This is an example of using the `ml` operator to get and set variables in the MATLAB workspace.



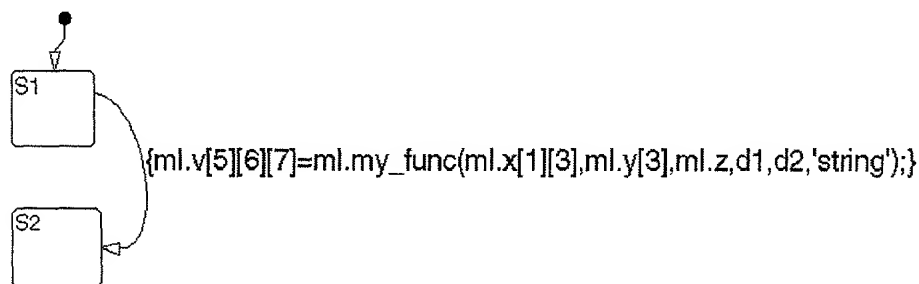
These data objects are defined in the Stateflow diagram:

- `d1` and `d2` are **Local** data objects
- `a`, `x`, and `y` must be defined in the MATLAB workspace prior to starting the simulation; otherwise a runtime error is generated at the execution time of the transition

The values of `a` and `y` are accessed in the MATLAB workspace and used in the expression with the **Local** data objects `d1` and `d2`. The result of the expression is assigned to the MATLAB workspace variable `x`. If `x` does not exist, it is automatically created in the MATLAB workspace.

Example: Using the ml Operator to Access MATLAB Functions

This is an example of using the `ml` operator to access MATLAB functions.



These data objects are defined:

- d1 and d2 are **Local** data objects defined in the Stateflow diagram
- x is assumed to be a two dimensional array in the MATLAB workspace
- y is assumed to be a MATLAB workspace vector.
- z is assumed to be a MATLAB workspace scalar variable.

x, y, and z must be defined in the MATLAB workspace prior to starting the simulation; otherwise a runtime error is generated at the execution time of the transition.

A MATLAB function named `my_func` is called with these arguments:

- 1 `x(1,3)`
- 2 `y(3)`
- 3 `z`
- 4 `d1`
- 5 `d2`
- 6 string `'abcdefgh'`

The result of `my_func()` (if it is a scalar) is assigned to element (5, 6, 7) of a multidimensional matrix `v` in the MATLAB workspace. If `v` does not exist prior to the execution of this statement, then it is automatically created by MATLAB workspace.

If `my_func()` returns a vector, the first element is assigned to `v(5,6,7)`. If it is a structure, a cell array, or a string, the result is undefined.

The `ml()` Function Versus `ml` Name Space Operator

It is recommended to use the `ml` name space operator wherever possible. The `ml` name space operator is faster and more robust than the `ml()` function. If you need to work with MATLAB matrices instead of scalars, then use the `ml()` function.

In this example, the `ml()` function must be used to specify an array argument.

```
a = ml('my_function([1:4],%g)',d1);
```

`x` is a MATLAB workspace matrix. `my_function` is a MATLAB function that expects a vector as its first argument and a scalar as a second argument.

Data and Event Arguments

Unqualified data and event objects are assumed to be defined at the same level in the hierarchy as the reference to them in the action language. Stateflow will attempt to resolve the object name by searching up the hierarchy. If the data or event object is parented elsewhere in the hierarchy, you need to define the hierarchy path explicitly.

Arrays

You can use arrays in the action language.

Examples of Array Assignments

Use C style syntax in the action language to access array elements.

```
local_array[1][8][0] = 10;
```

```
local_array[i][j][k] = 77;
```

```
var = local_array[i][j][k];
```

As an exception to this style, **scalar expansion** is available within the action language. This statement assigns a value of 10 to all of the elements of the array `local_array`.

```
local_array = 10;
```

Scalar expansion is available for performing general operations. This statement is valid if the arrays `array_1`, `array_2` and `array_3` have the same value for the **Sizes** property.

```
array_1 = (3*array_2) + array_3;
```

Using Arrays with Simulink

Array data objects that have a scope of **Input from Simulink** or **Output to Simulink** are constrained to one dimension. Use a single scalar value for the **Sizes** property of these arrays.

Arrays and Custom Code

The action language provides the same syntax for Stateflow arrays and custom code arrays. Any array variable that is referred to in a Stateflow chart but is not defined in the data dictionary is identified as a custom code variable.

Pointer and Address Operators

The Stateflow action language includes address and pointer operators. The address operator is available for use with both custom code variables and Stateflow variables. The pointer operator is available for use with custom code variables only.

Syntax Examples

These examples show syntax that is valid for use with *custom code* variables only.

```
varStruct.field = <expression>;
```

```
(*varPtr) = <expression>;
```

```
varPtr->field = <expression>;
```

```
myVar = varPtr->field;
```

```
varPtrArray[index]->field = <expression>;
```

```
varPtrArray[expression]->field = <expression>;
```

```
myVar = varPtrArray[expression]->field;
```

These examples show syntax that is valid for use with both custom code variables and Stateflow variables.

```
varPtr = &var;
```

```

ptr = &varArray[<expression>;

*(&var) = <expression>;

function(&varA, &varB, &varC);

function(&sf.varArray[<expr>]);

```

Syntax Error Detection

The action language parser uses a relaxed set of restrictions. As a result, many syntax errors will not be trapped until compilation.

Hexadecimal Notation

The action language supports C style hexadecimal notation (for example, 0xFF). You can use hexadecimal values wherever you can use decimal values.

Typecast Operators

A *typecast operator* converts a value to a specified data type. Stateflow typecast operators have the same notation as MATLAB typecast operators:

```
op(v)
```

where *op* is the typecast operator (e.g. int8, int16, int32, single, double) and *v* is the value to be converted.

Normally you do not need to use typecast operators in actions. This is because Stateflow checks whether the types involved in a variable assignment differ and, if so, inserts a typecast operator in the generated code. (Stateflow uses the typecast operator of the language in which the target is generated, typically C.) However, if external code defines either or both types, Stateflow cannot determine which typecast, if any, is required. If a type conversion is necessary, you must use a Stateflow action language typecast operator to tell Stateflow which target language typecast operator to generate.

For example, suppose *varA* is a data dictionary value of type *double* and *y* is an external variable of type 32-bit integer. The following notation

```
y = int32(varA)
```


tells Stateflow to generate a typecast operator that converts the value of `varA` to a 32-bit integer before the value is assigned to `y`.

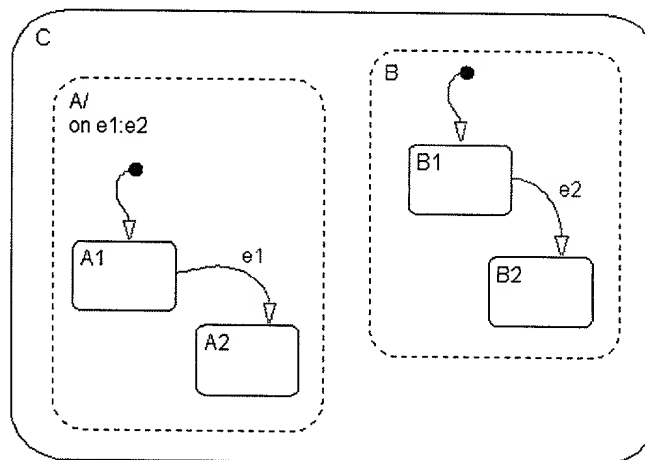
Event Broadcasting

You can specify an event to be broadcast in the action language. Events have hierarchy (a parent) and scope. The parent and scope together define a range of access to events. It is primarily the event's parent that determines who can trigger on the event (has receive rights). See “Name” on page 4-5 for more information.

Broadcasting an event in the action language is most useful as a means of synchronization amongst AND (parallel) states. Recursive event broadcasts can lead to definition of cyclic behavior. Cyclic behavior can be detected only during simulation.

Example: Event Broadcast State Action

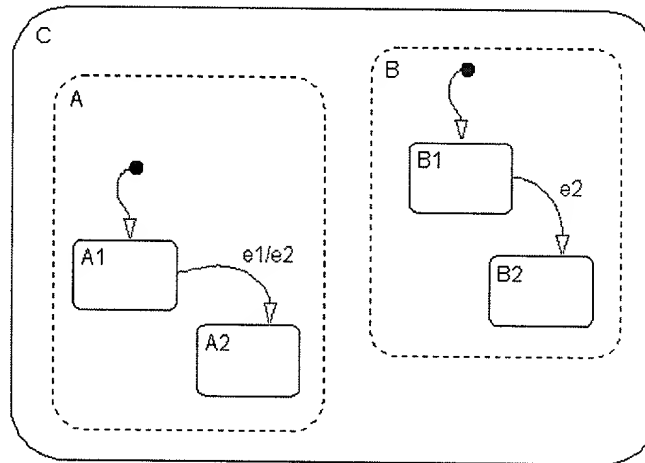
This is an example of the event broadcast state action notation.



See “Example: Event Broadcast State Action” on page 8-42 for information on the semantics of this notation.

Example: Event Broadcast Transition Action

This is an example of the event broadcast transition action notation.



See “Example: Event Broadcast Transition Action (Nested Event Broadcast)” on page 8-46 for information on the semantics of this notation.

Directed Event Broadcasting

You can specify a directed event broadcast in the action language. Using a directed event broadcast, you can broadcast a specific event to a specific receiver state. Directed event broadcasting is a more efficient means of synchronization amongst AND (parallel) states. Using directed event broadcasting improves the efficiency of the generated code. As is true in event broadcasting, recursive event broadcasts can lead to definition of cyclic behavior.

Note An action in one chart cannot broadcast events to states defined in another chart.

The format of the directed broadcast is

```
send(event_name, state_name)
```

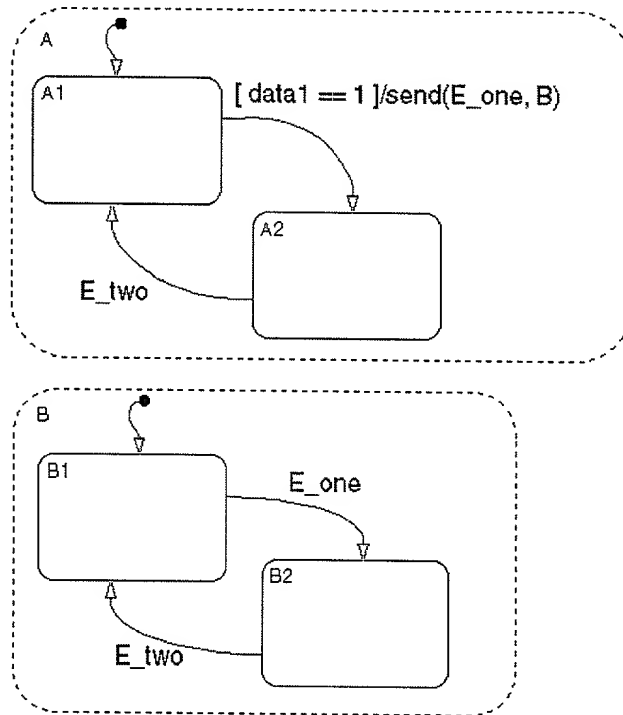
where event_name is broadcast to state_name (and any offspring of that state in the hierarchy). The state_name argument can include a full hierarchy path. For example,

```
send(event_name, chart_name.state_name1.state_name2)
```

The state_name specified must be active at the time the send is executed for the state_name to receive and potentially act on the directed event broadcast.

Example: Directed Event Broadcast Using send

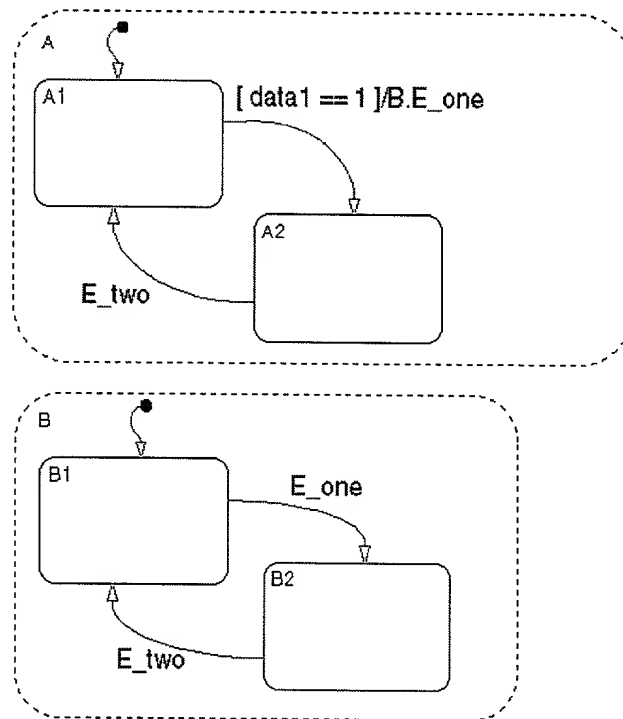
This is an example of a directed event broadcast using the `send(event_name, state_name)` transition action as a transition action.



In this example, event `E_one` must be visible in both A and B. See “Example: Directed Event Broadcasting Using Qualified Event Names” on page 8-56 for information on the semantics of this notation.

Example: Directed Event Broadcast Using Qualified Event Names

This example illustrates use of a qualified event name to in an event broadcast.



See “Example: Directed Event Broadcasting Using Qualified Event Names” on page 8-56 for information on the semantics of this notation.

Conditions

You sometimes want transitions or actions associated with transitions to take place only if a certain condition is true. Conditions are placed within `[]`. These are some guidelines for defining conditions:

- The expression must be a Boolean expression of some kind. The condition must evaluate to either true (1) or false(0).
- The expression can consist of:
 - Boolean operators that make comparisons between data and numeric values

- Any function that returns a Boolean value
- The `In(state_name)` condition function that is evaluated as true when the state specified as the argument is active. The full state name, including any ancestor states, must be specified to avoid ambiguity.
Note A chart cannot use the `In` condition function to trigger actions based on the activity of states in other charts.
- Temporal conditions (see “Temporal Logic Operators” on page 7-61)
- The condition expression should not call a function that causes the Stateflow diagram to change state or modify any variables.
- Boolean expressions can be grouped using `&` for expressions with AND relationships and `|` for expressions with OR relationships.
- Assignment statements are not valid condition expressions.
- Unary increment and decrement actions are not valid condition expressions.

Time Symbol

You can use the letter `t` to represent absolute time in simulation targets. This simulation time is inherited from Simulink.

For example, the condition `[t - On_time > Duration]` specifies that the condition is true if the value of `On_time` subtracted from the simulation time `t`, is greater than the value of `Duration`.

The meaning of `t` for nonsimulation targets is undefined since it is dependent upon the specific application and target hardware.

Literals

Place action language you want the parser to ignore but you want to appear as entered in the generated code within `$` characters. For example,

```
$
ptr -> field = 1.0;
$
```

The parser is completely disabled during the processing of anything between the `$` characters. Frequent use of literals is discouraged.

Continuation Symbols

Enter the characters ... at the end of a line to indicate the expression continues on the next line.

Comments

These comment formats are supported:

- % MATLAB comment line
- // C++ comment line
- /* C comment line */

Use of the Semicolon

Omitting the semicolon after an expression displays the results of the expression in the MATLAB command window. If you use a semicolon, the results are not displayed.

Temporal Logic Operators

Temporal logic operators are Boolean operators that operate on recurrence counts of Stateflow events. Stateflow defines the following temporal operators

- after
- before
- at
- every

The following sections explain the syntax and meaning of these operators and gives examples of their usage.

Usage Rules

The following rules apply generally to use of temporal logic operators.

- The recurring event on which a temporal operator operates is called the *base event*. Any Stateflow event can serve as a base event for a temporal operator. Note that temporal logic operators cannot operate on recurrences of implicit events, such as state entry or exit events.

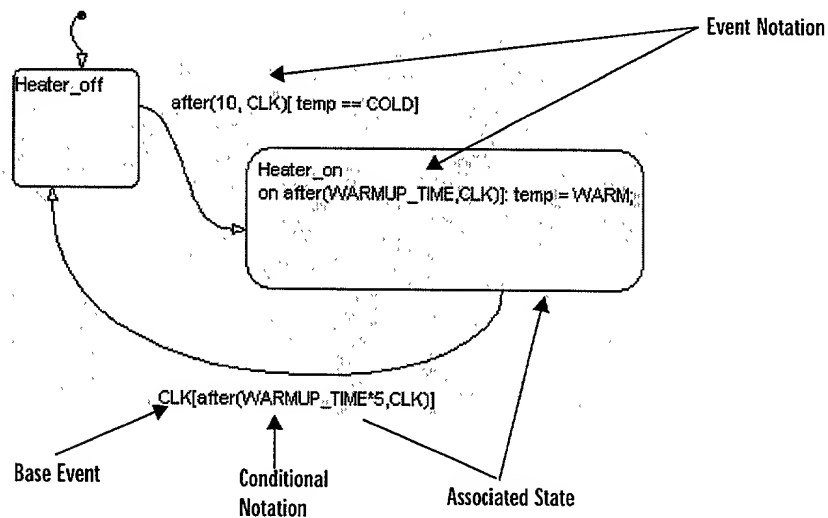
- Temporal logic operators can appear only in conditions on transitions originating from states and in state actions.

Note that this means you cannot use temporal logic operators as conditions on default transitions or flow graph transitions.

The state on which the temporally conditioned transition originates or in whose during action the condition appears is called the temporal operator's *associated state*.

- You must use event notation (see "Temporal Logic Events" on page 7-66) to express temporal logic conditions on events in state during actions.

The following diagram illustrates the usage and terminology that apply to temporal logic operators.



After Operator

Syntax

after (n, E)

where E is the base event for the operator and n is any expression that evaluates to a positive integer value.

Semantics

The after operator is true if the base event E has occurred *n* times since the operator's associated state was activated. Otherwise, it is false.

Note The after operator resets its counter for E to 0 each time the associated state is activated.

Usage

The following example illustrate use of the after operator in a transition expression.

```
CLK[after(10, CLK) && temp == COLD]
```

This example permits a transition out of the associated state only if there have been 10 occurrences of the CLK event since the state was activated and the temp data item has the value COLD.

The next example illustrates usage of event notation for temporal logic conditions in transition expressions.

```
after(10, CLK) [temp == COLD]
```

This example is semantically equivalent to the first example.

The next example illustrates setting a transition condition for any event visible in the associated state while it is activated.

```
[after(10, CLK)]
```

This example permits a transition out of the associated state on any event after 10 occurrences of the CLK event since activation of the state.

The next two examples underscore the semantic distinction between an after condition on its own base event and an after condition on a nonbase event.

```
CLK[after(10, CLK)]  
ROTATE[after(10, CLK)]
```

The first expression says that the transition must occur *as soon as* 10 CLK events have occurred after activation of the associated state. The second expression says that the transition may occur *no sooner than* 10 CLK events

after activation of the state, but possibly later, depending on when the ROTATION event occurs.

The next example illustrates usage of an after event in a state's during action.

```
Heater_on  
on after (5*BASE_DELAY, CLK): status('heater on');
```

This example causes the Heater_on state to display a status message each CLK cycle, starting 5*BASE_DELAY clock cycles after activation of the state. Note the use of event notation to express the after condition in this example. Use of conditional notation is not allowed in state during actions.

Before Operator

Syntax

before(n, E)

where E is the base event for the operator and n is any expression that evaluates to a positive integer value.

Semantics

The before operator is true if the base event E has occurred less than n times since the operator's associated state was activated. Otherwise, it is false.

Note The before operator resets its counter for E to 0 each time the associated state is activated.

Usage

The following example illustrate use of the before operator in a transition expression.

```
ROTATION[before(10, CLK)]
```

This expression permits a transition out of the associated state only on occurrence of a ROTATION event but *no later than* 10 CLK cycles after activation of the state.

The next example illustrates usage of a before event in a state's during action.

```
Heater_on  
on before(MAX_ON_TIME, CLK): temp++;
```

This example causes the Heater_on state to increment the temp variable once per CLK cycle until the MAX_ON_TIME limit is reached.

At Operator

Syntax

at(n, E)

where E is the base event for the at operator and n is any expression that evaluates to an integer value.

Semantics

The at operator is true only at the nth occurrence of the base event E since activation of the associated state.

Note The at operator resets its counter for E to 0 each time the associated state is activated.

Usage

The following example illustrate use of the at operator in a transition expression.

```
ROTATION[at(10, CLK)]
```

This expression permits a transition out of the associated state only if a ROTATION event occurs *exactly* 10 CLK cycles after activation of the state.

The next example illustrates usage of a before event in a state's during action.

```
Heater_on  
on at(10, CLK): status("heater on");
```

This example causes the Heater_on state to display a status message 10 CLK cycles after activation of the associated state.

Every Operator

Syntax

`every(n, E)`

where E is the base event for the at operator and n is any expression that evaluates to an integer value.

Semantics

The at operator is true at every nth occurrence of the base event E since activation of the associated state.

Note The every operator resets its counter for E to 0 each time the associated state is activated. As a result, this operator is useful only in state during actions.

Usage

The following example illustrate use of the at operator in a state during.

```
Heater_on
on every(10, CLK): status("heater on");
```

This example causes the Heater_on state to display a status message every 10 CLK cycles after activation of the associated state.

Temporal Logic Events

Stateflow treats the following notations as equivalent

```
E[to(n, E) && C]
to(n, E)[C]
```

where to is a temporal operator (after, before, at, every), E is the operator's base event, n is the operator's occurrence count, and C is any conditional expression. For example, the following expressions are functionally equivalent in Stateflow.

```
CLK[after(10, CLK) && temp == COLD]
after(10, CLK)[temp == COLD]
```

The first notation is referred to as the conditional notation for temporal logic operators and the second notation as the event notation.

Note You can use conditional and event notation interchangeably in transition expressions. However, you must use the event notation in state during actions.

Although temporal logic does not introduce any new events into a Stateflow model, it is useful to think of the change of value of a temporal logic condition as an event. For example, suppose that you want a transition to occur from state A exactly 10 clock cycles after activation of the state. One way to achieve this would be to define an event called ALARM and to broadcast this event 10 CLK events after state A is entered. You would then use ALARM as the event that triggers the transition out of state A.

An easier way to achieve the same behavior is to set a temporal logic condition on the CLK event that triggers the transition out of state A.

```
CLK[after(10, CLK)]
```

Note that this approach does not require creation of any new events. Nevertheless, conceptually it is useful to think of this expression as equivalent to creation of an implicit event that triggers the transition. Hence, Stateflow's support for the equivalent event notation.

```
after(10, CLK)
```

Note that the event notation allows you to set additional constraints on the implicit temporal logic "event," for example,

```
after(10, CLK) [temp == COLD]
```

This expression says, "Exit state A if the temperature is cold but no sooner than 10 clock cycles."

7 Notations

7-68



Semantics

Overview	8-2
Event-Driven Effects on Semantics	8-5
Transitions to and from Exclusive (OR) States	8-8
Condition Actions	8-13
Default Transitions	8-18
Inner Transitions	8-23
Connective Junctions	8-31
Event Actions	8-40
Parallel (AND) States	8-42
Directed Event Broadcasting	8-54
Execution Order	8-58
Semantic Rules Summary	8-62

Overview

Semantics describe how the notation is interpreted and implemented. A completed Stateflow diagram communicates how the system will behave. A Stateflow diagram contains actions associated with transitions and states. The semantics describe in what sequence these actions take place during Stateflow diagram execution.

Knowledge of the semantics is important to make sound Stateflow diagram design decisions for code generation. Different use of notations results in different ordering of simulation and generated code execution.

Stateflow semantics consist of rules for:

- Event broadcasting
- Processing states
- Processing transitions
- Taking transition paths

The details of Stateflow semantics are described largely by examples in this chapter. The examples cover a range of various notations and combinations of state and transition actions.

See “Semantic Rules Summary” on page 8-62 for a summary of the semantics.

List of Semantic Examples

This is a list of the semantic examples provided in this chapter.

Transitions to and from Exclusive (OR) States

- “Example: Processing of One Event” on page 8-8
- “Example: Processing of a Second Event” on page 8-9
- “Example: Processing of a Third Event” on page 8-10
- “Example: Transition from a Substate to a Substate” on page 8-11

Condition Actions

- “Example: Actions Specified as Condition Actions” on page 8-13

- “Example: Actions Specified as Condition and Transition Actions” on page 8-14
- “Example: Using Condition Actions in For Loop Construct” on page 8-15
- “Example: Using Condition Actions to Broadcast Events to Parallel (AND) States” on page 8-16
- “Example: Cyclic Behavior to Avoid When Using Condition Actions” on page 8-17

Default Transitions

- “Example: Default Transition in an Exclusive (OR) Decomposition” on page 8-18
- “Example: Default Transition to a Junction” on page 8-19
- “Example: Default Transition and a History Junction” on page 8-20
- “Example: Labeled Default Transitions” on page 8-21

Inner Transitions

- “Example: Processing One Event Within an Exclusive (OR) State” on page 8-23
- “Example: Processing a Second Event Within an Exclusive (OR) State” on page 8-24
- “Example: Processing a Third Event Within an Exclusive (OR) State” on page 8-25
- “Example: Processing One Event with an Inner Transition to a Connective Junction” on page 8-26
- “Example: Processing a Second Event with an Inner Transition to a Connective Junction” on page 8-27
- “Example: Inner Transition to a History Junction” on page 8-29

Connective Junctions

- “Example: If-Then-Else Decision Construct” on page 8-31
- “Example: Self Loop” on page 8-32
- “Example: For Loop Construct” on page 8-33
- “Example: Flow Diagram Notation” on page 8-34

- “Example: Transitions from a Common Source to Multiple Destinations” on page 8-36
- “Example: Transitions from Multiple Sources to a Common Destination” on page 8-37
- “Example: Transitions from a Source to a Destination Based on a Common Event” on page 8-38

Event Actions

- “Example: Event Actions and Superstates” on page 8-40

Parallel (AND) States

- “Example: Event Broadcast State Action” on page 8-42
- “Example: Event Broadcast Transition Action (Nested Event Broadcast)” on page 8-46
- “Example: Event Broadcast Condition Action” on page 8-50

Directed Event Broadcasting

- “Example: Directed Event Broadcast Using send” on page 8-54
- “Example: Directed Event Broadcasting Using Qualified Event Names” on page 8-56

Event-Driven Effects on Semantics

What Does Event-Driven Mean?

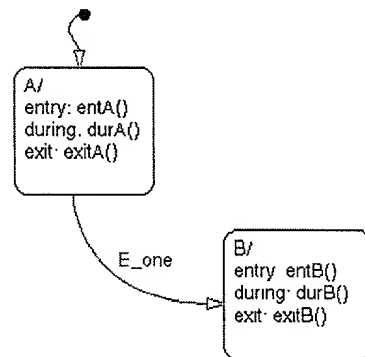
The Stateflow diagram executes only when an event occurs; an event occurs and the Stateflow diagram is awakened to respond to the event. Exactly what executes depends on the circumstances when the event occurs. Actions that are to take place based on an event are atomic to that event. Once an action is initiated, it is completed unless interrupted by an early return.

Top-Down Processing of Events

When an event occurs, it is processed from the top or root of the Stateflow diagram down through the hierarchy of the Stateflow diagram. At each level in the hierarchy, any during and on *event_name* actions for the active state are executed and completed and then a check for the existence of a valid explicit or implicit transition among the children of the state is conducted. The examples in this chapter demonstrate the top-down processing of events.

Semantics of Active and Inactive States

This example shows the semantics of active and inactive states.



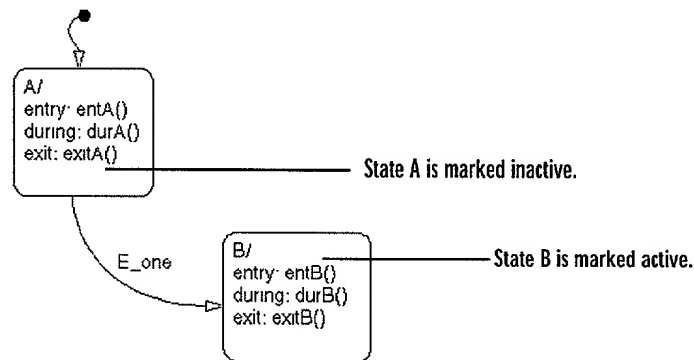
Initially the Stateflow diagram is asleep and both states are inactive. An event occurs and the Stateflow diagram is awakened. This is the semantic sequence:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of the event. A valid default transition to state A is detected.

- 2 State A is marked active.
- 3 State A entry actions execute and complete (entA()).
- 4 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

Event E_one occurs and the Stateflow diagram is awakened. State A is active. This is the semantic sequence:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. A valid transition is detected from state A to state B.
- 2 State A exit actions execute and complete (exitA()).
- 3 State A is marked inactive.
- 4 State B is marked active.
- 5 State B entry actions execute and complete (entB()).
- 6 The Stateflow diagram goes back to sleep, to be awakened by the next event.



Semantics of State Actions

An entry action is executed as a result of any transition into the state. The state is marked active before its entry action is executed and completed.

A during action executes to completion when that state is active and an event occurs that does not result in an exit from that state. An *on event_name* action executes to completion when the event specified, *event_name*, occurs and that state is active. An active state executes its during and *on event_name* actions before processing any of its children's valid transitions. During and *on event_name* actions are processed based on their order of appearance in the state label.

An exit action is executed as a result of any transition out of the state. The state is marked inactive after the exit action has executed and completed.

Semantics of Transitions

Transitions play a large role in defining the animation or execution of a system. Transitions have sources and destinations; thus any actions associated with the sources or destinations are related to the transition that joins them. The type of the source and destination is equally important to define the semantics.

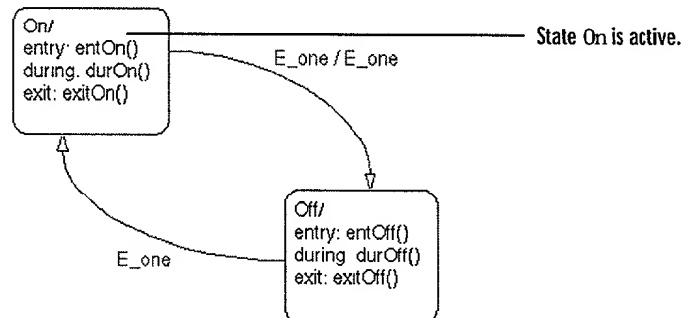
The examples provided in this chapter show how the semantics are defined.

Figure 8-7: State Machine Diagrams for the Examples in This Chapter

Transitions to and from Exclusive (OR) States

Example: Processing of One Event

This example shows the semantics of a simple transition focusing on the implications of states being active or inactive.



Initially the Stateflow diagram is asleep. State **On** and state **Off** are OR states. State **On** is active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

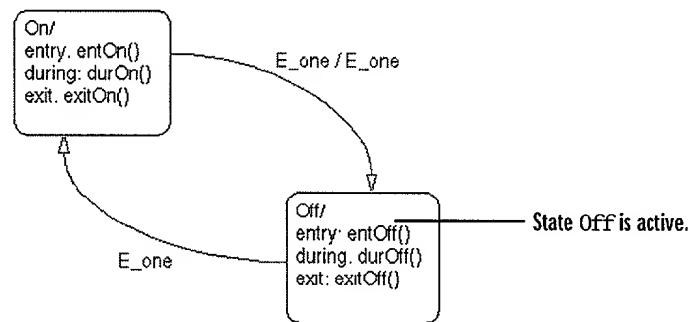
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. A valid transition from state **On** to state **Off** is detected.
- 2 State **On** exit actions execute and complete (`ExitOn()`).
- 3 State **On** is marked inactive.
- 4 The event `E_one` is broadcast as the transition action. The second generation of event `E_one` is processed but because neither state is active, it has no effect. (Had a valid transition been possible as a result of the broadcast of `E_one`, the processing of the first broadcast of `E_one` would be preempted by the second broadcast of `E_one`.)
- 5 State **Off** is marked active.
- 6 State **Off** entry actions execute and complete (`entOff()`).

- 7 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of the Stateflow diagram associated with event E_one when state On was active.

Example: Processing of a Second Event

Using the same example, what happens when the next event, E_one, occurs?



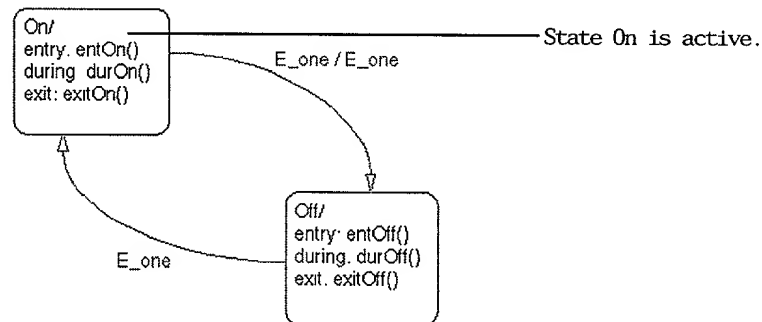
Again, initially the Stateflow diagram is asleep. State Off is active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. A valid transition from state Off to state On is detected.
- 2 State Off exit actions execute and complete (exitOff()).
- 3 State Off is marked inactive.
- 4 State On is marked active.
- 5 State On entry actions execute and complete (entOn()).
- 6 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of the Stateflow diagram associated with the second event E_one when state Off was active.

Example: Processing of a Third Event

Using the same example, what happens when a third event, E_two, occurs?



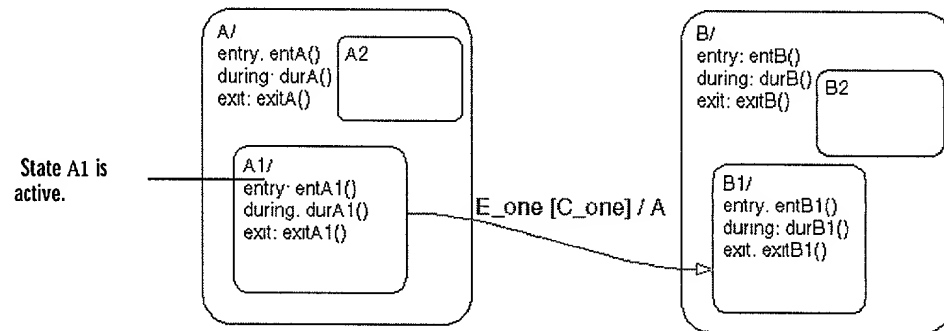
Again, initially the Stateflow diagram is asleep. State On is active. Event E_two occurs and awakens the Stateflow diagram. Event E_two is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_two. There is none.
- 2 State On during actions execute and complete (durOn()).
- 3 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of the Stateflow diagram associated with event E_two when State On was active.

Example: Transition from a Substate to a Substate

This example shows the semantics of a transition from an OR substate to an OR substate.



Initially the Stateflow diagram is asleep. State A.A1 is active. Event `E_one` occurs and awakens the Stateflow diagram. Condition `C_one` is true. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is a valid transition from state A.A1 to state B.B1. (Condition `C_one` is true.)
- 2 State A executes and completes during actions (`durA()`).
- 3 State A.A1 executes and completes exit actions (`exitA1()`).
- 4 State A.A1 is marked inactive.
- 5 State A executes and completes exit actions (`exitA()`).
- 6 State A is marked inactive.
- 7 The transition action, A, is executed and completed.
- 8 State B is marked active.
- 9 State B executes and completes entry actions (`entB()`).
- 10 State B.B1 is marked active.

11 State B.B1 executes and completes entry actions (entB1()).

12 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

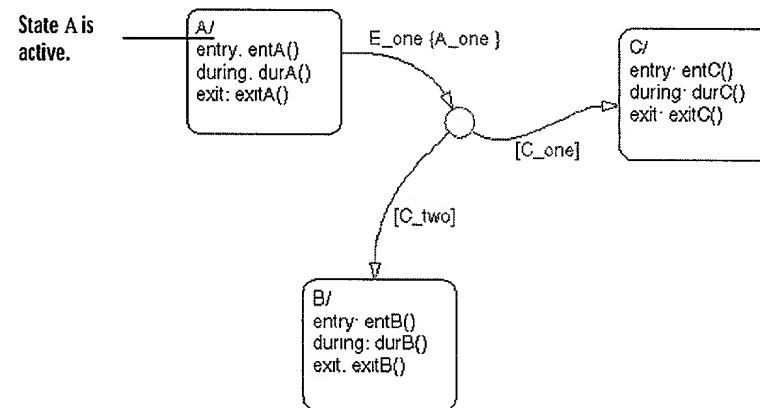
This sequence completes the execution of this Stateflow diagram associated with event E_one.

Stateflow diagram associated with event E_one.

Condition Actions

Example: Actions Specified as Condition Actions

This example shows the semantics of a simple condition action in a multiple segment transition.



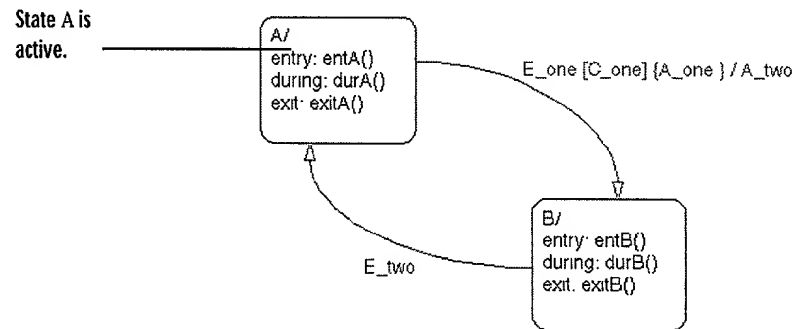
Initially the Stateflow diagram is asleep. State A is active. Event `E_one` occurs and awakens the Stateflow diagram. Conditions `C_one` and `C_two` are false. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. A valid transition segment from state A to a connective junction is detected. The condition action, `A_one`, is detected on the valid transition segment and is immediately executed and completed. State A is still active.
- 2 Since the conditions on the transition segments to possible destinations are false, none of the complete transitions is valid.
- 3 State A remains active. State A during action executes and completes (`durA()`).
- 4 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of the Stateflow diagram associated with event E_one when state A was active.

Example: Actions Specified as Condition and Transition Actions

This example shows the semantics of a simple condition and transition action specified on a transition from one exclusive (OR) state to another.



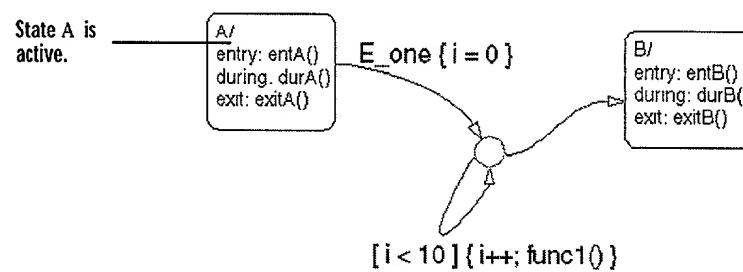
Initially the Stateflow diagram is asleep. State A is active. Event E_one occurs and awakens the Stateflow diagram. Condition C_one is true. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. A valid transition from state A to state B is detected. The condition, C_one is true. The condition action, A_one, is detected on the valid transition and is immediately executed and completed. State A is still active.
- 2 State A exit actions execute and complete (ExitA()).
- 3 State A is marked inactive.
- 4 The transition action, A_two, is executed and completed.
- 5 State B is marked active.
- 6 State B entry actions execute and complete (entB()).
- 7 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of the Stateflow diagram associated with event E_one when state A was active.

Example: Using Condition Actions in For Loop Construct

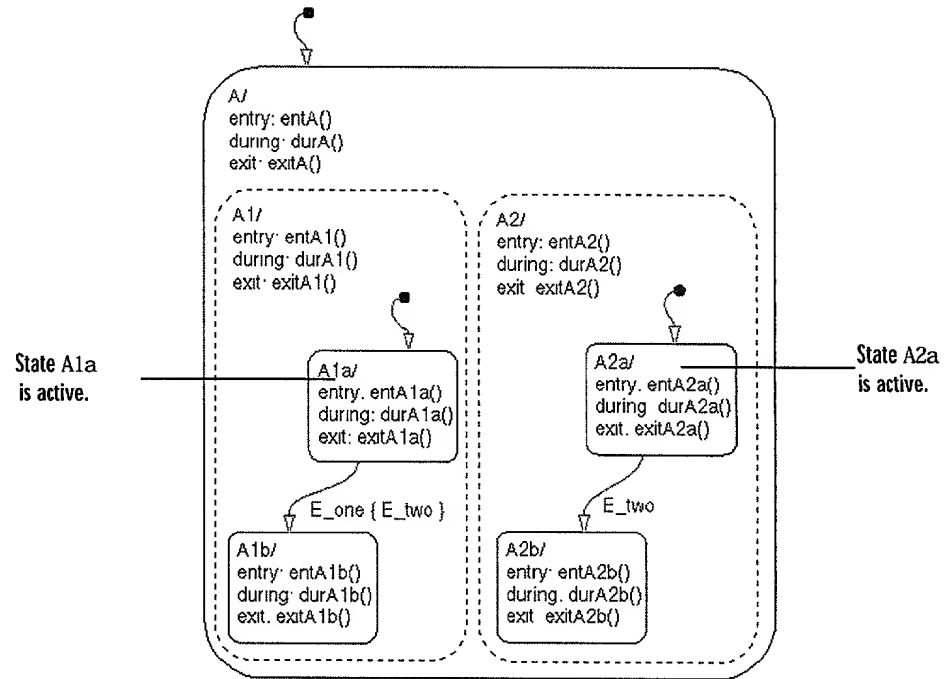
Condition actions and connective junctions are used to design a for loop construct. This example shows the use of a condition action and connective junction to create a for loop construct.



See “Example: For Loop Construct” on page 8-33 to see the semantics of this example.

Example: Using Condition Actions to Broadcast Events to Parallel (AND) States

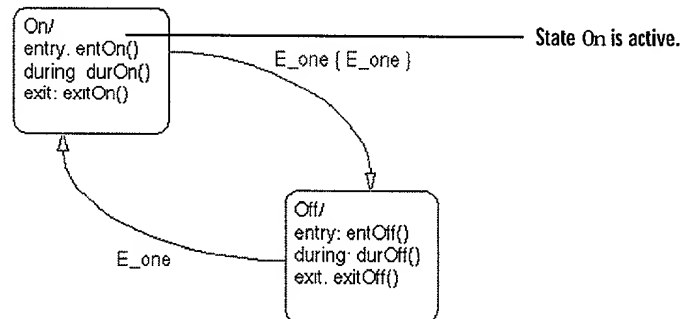
Condition actions can be used to broadcast events immediately to parallel (AND) states. This example shows this use.



See "Example: Event Broadcast Condition Action" on page 8-50 to see the semantics of this example.

Example: Cyclic Behavior to Avoid When Using Condition Actions

This example shows a notation to avoid when using event broadcasts as condition actions because the semantics result in cyclic behavior.



Initially the Stateflow diagram is asleep. State On is active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. A valid transition from state On to state Off is detected. A condition action, broadcast of event `E_one`, is detected on the valid transition and is immediately executed. State On is still active.

The broadcast of event `E_one` awakens the Stateflow diagram a second time. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. The transition from state On to state Off is still valid. The condition action, broadcast of event `E_one`, is immediately executed again.

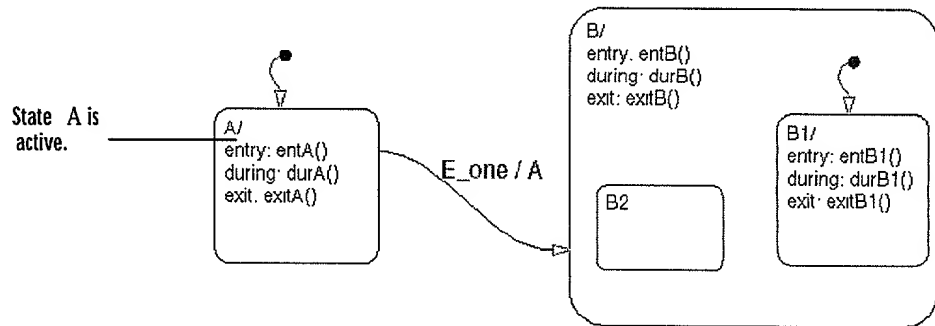
- 2 Step 1 continues to execute in a cyclical manner. The transition label indicating a trigger on the same event as the condition action broadcast event results in unrecoverable cyclic behavior.

This sequence never completes when event `E_one` is broadcast and state On is active.

Default Transitions

Example: Default Transition in an Exclusive (OR) Decomposition

This example shows a transition from an OR state to a superstate with exclusive (OR) decomposition, where a default transition to a substate is defined.



Initially the Stateflow diagram is asleep. State A is active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

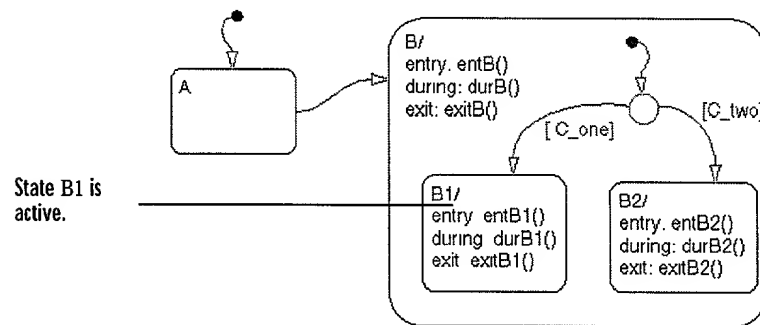
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition from state A to superstate B.
- 2 State A exit actions execute and complete (exitA()).
- 3 State A is marked inactive.
- 4 The transition action, A, is executed and completed.
- 5 State B is marked active.
- 6 State B entry actions execute and complete (entB()).
- 7 State B detects a valid default transition to state B.B1.
- 8 State B.B1 is marked active.
- 9 State B.B1 entry actions execute and complete (entB1()).

- 10 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Example: Default Transition to a Junction

This example shows the semantics of a default transition to a connective junction.



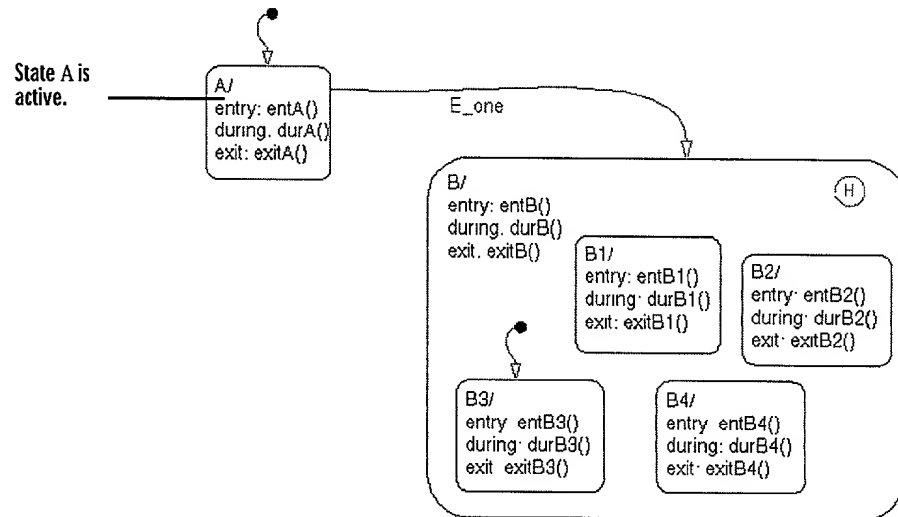
Initially the Stateflow diagram is asleep. State B.B1 is active. An event occurs and awakens the Stateflow diagram. Condition [C_two] is true. The event is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 State B checks to see if there is a valid transition as a result of any event. There is none.
- 2 State B1 during actions execute and complete (durB1()).

This sequence completes the execution of this Stateflow diagram associated with the occurrence of any event.

Example: Default Transition and a History Junction

This example shows the semantics of a superstate and a history junction.



Initially the Stateflow diagram is asleep. State A is active. There is a history junction and state B4 was the last active substate of superstate B. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

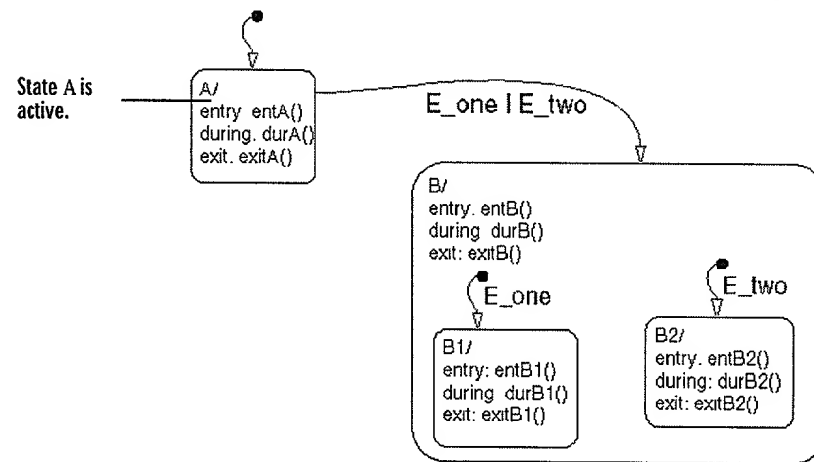
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is valid transition from state A to superstate B.
- 2 State A exit actions execute and complete (`exitA()`).
- 3 State A is marked inactive.
- 4 State B is marked active.
- 5 State B entry actions execute and complete (`entB()`).
- 6 State B detects and uses the history junction to determine which substate is the destination of the transition into the superstate. The history junction indicates substate B.B4 was the last active substate, and thus the destination of the transition.

- 7 State B.B4 is marked active.
- 8 State B.B4 entry actions execute and complete (entB4()).
- 9 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Example: Labeled Default Transitions

This example shows the use of a default transition with a label.



Initially the Stateflow diagram is asleep. State A is active. Event E_one occurs awakening the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

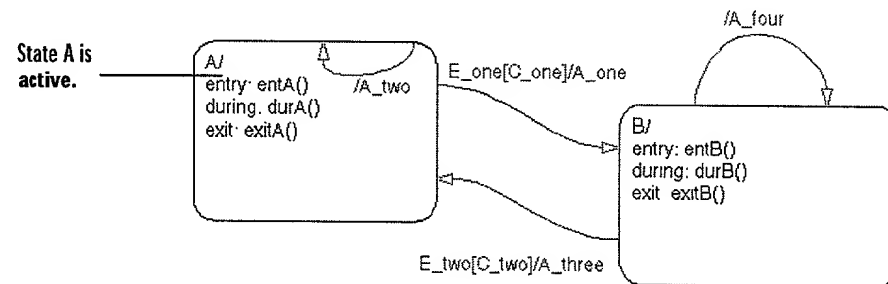
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition from state A to superstate B. A pipe is used to represent that the transition is valid if event E_one or E_two occurs.
- 2 State A exit actions execute and complete (exitA()).
- 3 State A is marked inactive.
- 4 State B is marked active.
- 5 State B entry actions execute and complete (entB()).
- 6 State B detects a valid default transition to state B.B1. The default transition is valid as a result of E_one.
- 7 State B.B1 is marked active.
- 8 State B.B1 entry actions execute and complete (entB1()).
- 9 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Inner Transitions

Example: Processing One Event Within an Exclusive (OR) State

This example shows the semantics of an inner transition.



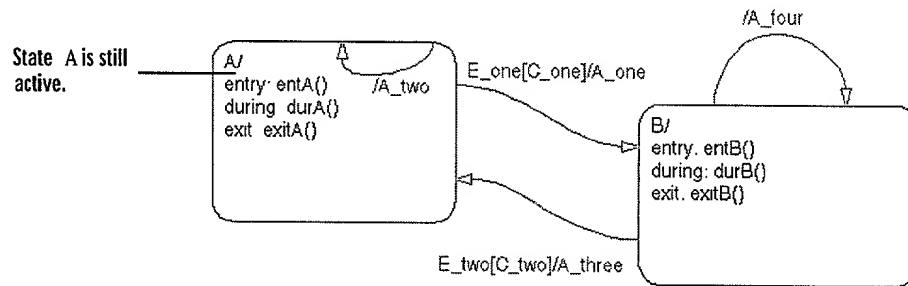
Initially the Stateflow diagram is asleep. State A is active. Event `E_one` occurs and awakens the Stateflow diagram. Condition `[C_one]` is false. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. A potentially valid transition from state A to state B is detected. However the transition is not valid because `[C_one]` is false.
- 2 State A during actions execute and complete (`durA()`).
- 3 State A checks its children for a valid transition and detects a valid inner transition.
- 4 State A remains active. The inner transition action, `A_two`, is executed and completed. Because it is an inner transition, state A's exit and entry actions are not executed.
- 5 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one`.

Example: Processing a Second Event Within an Exclusive (OR) State

Using the same example, what happens when a second event, E_one, occurs?



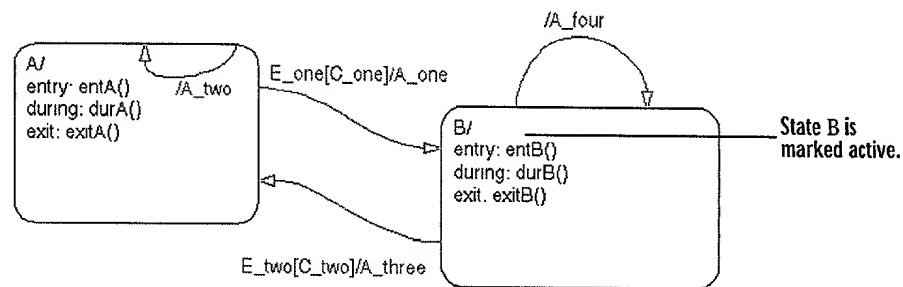
Initially the Stateflow diagram is asleep. State A is still active. Event E_one occurs and awakens the Stateflow diagram. Condition [C_one] is true. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. The transition from state A to state B is now valid because [C_one] is true.
- 2 State A exit actions execute and complete (exitA()).
- 3 State A is marked inactive.
- 4 The transition action A_one is executed and completed.
- 5 State B is marked active.
- 6 State B entry actions execute and complete (entB()).
- 7 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Example: Processing a Third Event Within an Exclusive (OR) State

Using the same example, what happens when a third event, E_two, occurs?



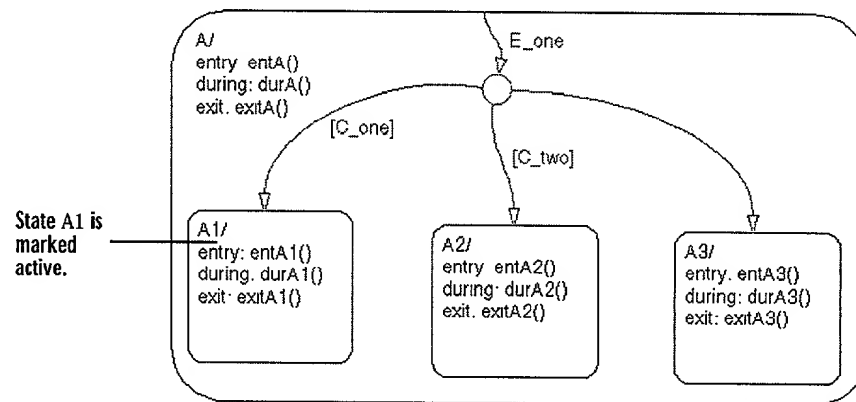
Initially the Stateflow diagram is asleep. State B is now active. Event E_two occurs and awakens the Stateflow diagram. Condition [C_two] is false. Event E_two is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_two. A potentially valid transition from state B to state A is detected. The transition is not valid because [C_two] is false. However, active state B has a valid self loop transition.
- 2 State B exit actions execute and complete (exitB()).
- 3 State B is marked inactive.
- 4 The self loop transition action, A_four, executes and completes.
- 5 State B is marked active.
- 6 State B entry actions execute and complete (entB()).
- 7 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_two. This example shows the difference in semantics between inner transitions and self loop transitions.

Example: Processing One Event with an Inner Transition to a Connective Junction

This example shows the semantics of an inner transition to a connective junction.



Initially the Stateflow diagram is asleep. State A1 is active. Event E_one occurs and awakens the Stateflow diagram. Condition [C_two] is true. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

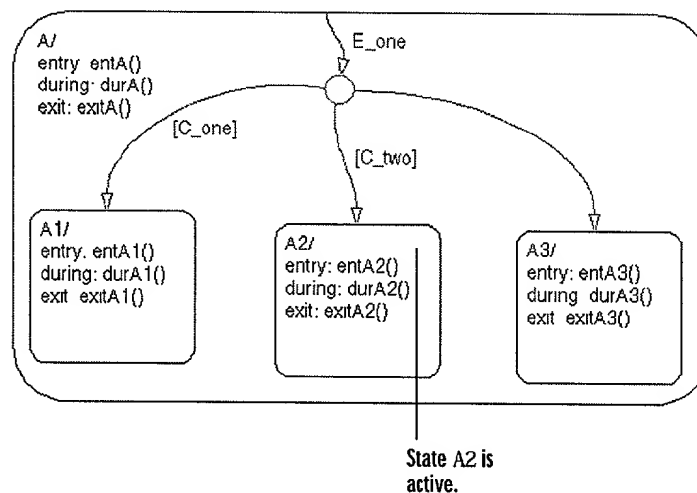
- 1 The Stateflow diagram root checks to see if there is a valid transition at the root level, as a result of E_one. There is no valid transition.
- 2 State A during actions execute and complete (durA()).
- 3 State A checks itself for valid transitions and detects there is a valid inner transition to a connective junction. The conditions are evaluated to determine if one of the transitions is valid. The segments labeled with a condition are evaluated before the unlabeled segment. The evaluation starts from a twelve o'clock position on the junction and progresses in a clockwise manner. Since [C_two] is true, the inner transition to the junction and then to state A.A2 is valid.

- 4 State A.A1 exit actions execute and complete (exitA1()).
- 5 State A.A1 is marked inactive.
- 6 State A.A2 is marked active.
- 7 State A.A2 entry actions execute and complete (entA2()).
- 8 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one when condition C_two is true.

Example: Processing a Second Event with an Inner Transition to a Connective Junction

This example shows the semantics of an inner transition to a junction when a second event, E_one, occurs.



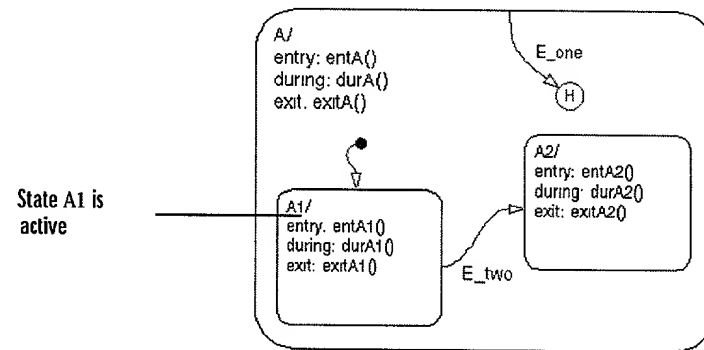
Initially the Stateflow diagram is asleep. State A2 is active. Event E_one occurs and awakens the Stateflow diagram. Neither `[C_one]` nor `[C_two]` is true. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition at the root level, as a result of E_one. There is no valid transition.
- 2 State A during actions execute and complete (durA()).
- 3 State A checks itself for valid transitions and detects a valid inner transition to a connective junction. The segments labeled with a condition are evaluated before the unlabeled segment. The evaluation starts from a twelve o'clock position on the junction and progresses in a clockwise manner. Since neither [C_one] nor [C_two] is true, the unlabeled transition segment is evaluated and is determined to be valid. The full transition from the inner transition to state A.A3 is valid.
- 4 State A.A2 exit actions execute and complete (exitA2()).
- 5 State A.A2 is marked inactive.
- 6 State A.A3 is marked active.
- 7 State A.A3 entry actions execute and complete (entA3()).
- 8 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one when neither [C_one] nor [C_two] is true.

Example: Inner Transition to a History Junction

This example shows the semantics of an inner transition to a history junction.



Initially the Stateflow diagram is asleep. State A.A1 is active. There is history information since superstate A is active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is no valid transition.
- 2 State A during actions execute and complete (durA()).
- 3 State A checks itself for valid transitions and detects there is a valid inner transition to a history junction. According to the semantics of history junctions, the last active state, A.A1, is the destination state.
- 4 State A.A1 exit actions execute and complete (exitA1()).
- 5 State A.A1 is marked inactive.
- 6 State A.A1 is marked active.
- 7 State A.A1 entry actions execute and complete (entA1()).
- 8 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

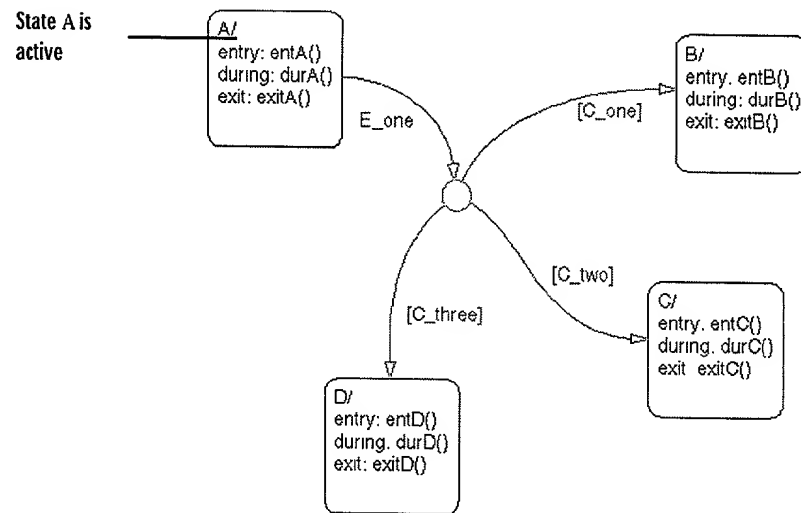
This sequence completes the execution of this Stateflow diagram associated with event E_one when there is an inner transition to a history junction and state A.A1 is active.

Stateflow diagram associated with event E_one when there is an inner transition to a history junction and state A.A1 is active.

Connective Junctions

Example: If-Then-Else Decision Construct

This example shows the semantics of an if-then-else decision construct.



Initially the Stateflow diagram is asleep. State A is active. Event E_one occurs and awakens the Stateflow diagram. Condition [C_two] is true. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

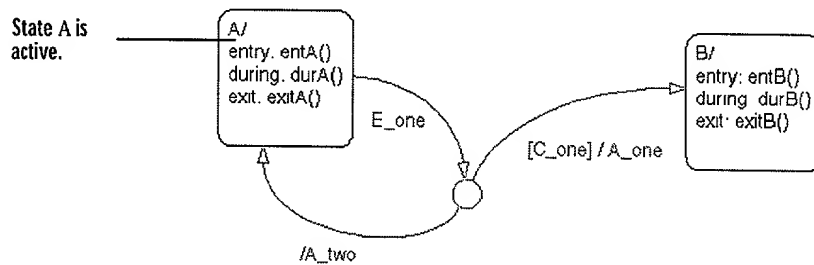
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition segment from state A to the connective junction. The transition segments beginning from a twelve o'clock position on the connective junction are evaluated for validity. The first transition segment labeled with condition [C_one] is not valid. The next transition segment labeled with the condition [C_two] is valid. The complete transition from state A to state C is valid.
- 2 State A executes and completes exit actions (exitA()).
- 3 State A is marked inactive.
- 4 State C is marked active.

- 5 State C executes and completes entry actions (`entC()`).
- 6 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one`.

Example: Self Loop

This example shows the semantics of a self loop using a connective junction.



Initially the Stateflow diagram is asleep. State A is active. Event `E_one` occurs and awakens the Stateflow diagram. Condition `[C_one]` is false. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

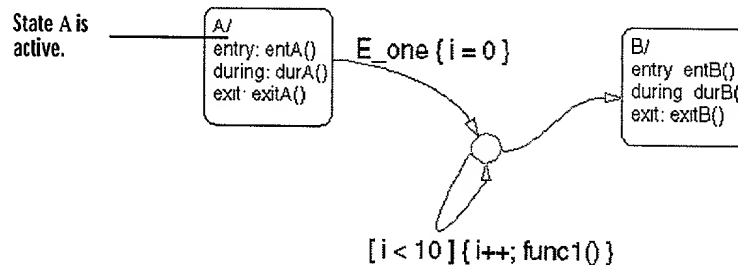
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is a valid transition segment from state A to the connective junction. The transition segment labeled with a condition and action is evaluated for validity. Since the condition `[C_one]` is not valid, the complete transition from state A to state B is not valid. The transition segment from the connective junction back to state A is valid.
- 2 State A executes and completes exit actions (`exitA()`).
- 3 State A is marked inactive.
- 4 The transition action `A_two` is executed and completed.
- 5 State A is marked active.

- 6 State A executes and completes entry actions (entA()).
- 7 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Example: For Loop Construct

This example shows the semantics of a for loop.



Initially the Stateflow diagram is asleep. State A is active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

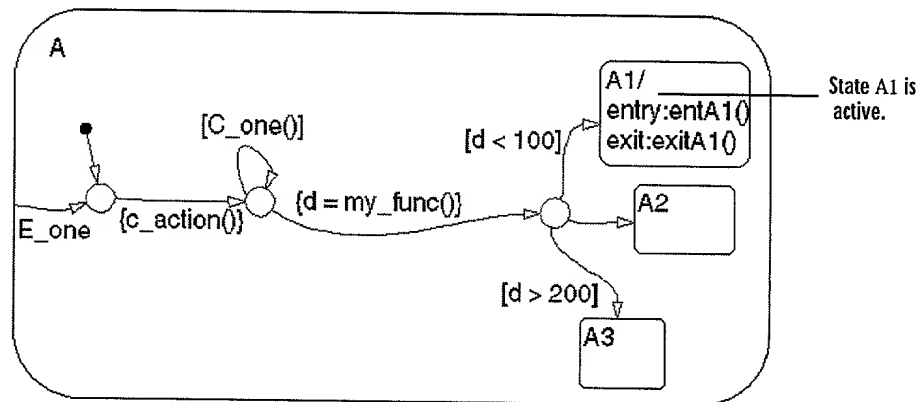
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition segment from state A to the connective junction. The transition segment condition action, $i = 0$, is executed and completed. Of the two transition segments leaving the connective junction, the transition segment that is a self loop back to the connective junction is evaluated next for validity. That segment takes priority in evaluation because it has a condition specified whereas the other segment is unlabeled.
- 2 The condition $[i < 10]$ is evaluated as true. The condition actions, $i++$, and a call to `func1` are executed and completed until the condition becomes false. A connective junction is not a final destination; thus the transition destination remains to be determined.

- 3 The unconditional segment to state B is now valid. The complete transition from state A to state B is valid.
- 4 State A executes and completes exit actions (exitA()).
- 5 State A is marked inactive.
- 6 State B is marked active.
- 7 State B executes and completes entry actions (entB()).
- 8 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Example: Flow Diagram Notation

This example shows the semantics of a Stateflow diagram that uses flow notation.



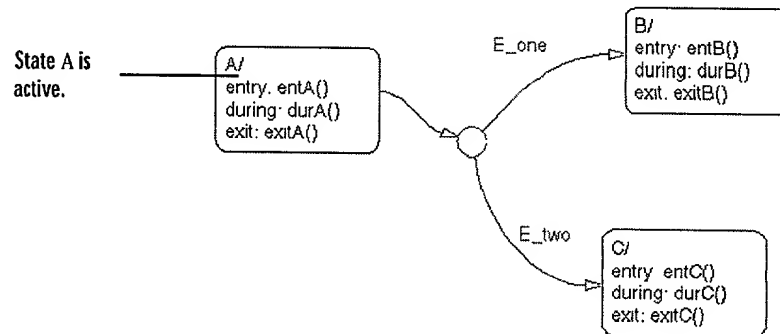
Initially the Stateflow diagram is asleep. State A.A1 is active. The condition [C_one()] is initially true. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is no valid transition.
- 2 State A checks itself for valid transitions and detects a valid inner transition to a connective junction.
- 3 The next possible segments of the transition are evaluated. There is only one outgoing transition and it has a condition action defined. The condition action is executed and completed.
- 4 The next possible segments are evaluated. There are two outgoing transitions; one is a conditional self loop and the other is an unconditional transition segment. The conditional transition segment takes precedence. The condition [C_one()] is tested and is true; the self loop is taken. Since a final transition destination has not been reached, this self loop continues until [C_one()] is false. Assume that after five loops [C_one()] is false.
- 5 The next possible transition segment (to the next connective junction) is evaluated. It is an unconditional transition segment with a condition action. The transition segment is taken and the condition action, {d=my_func() }, is executed and completed. The returned value of d is 84.
- 6 The next possible transition segment is evaluated. There are three possible outgoing transition segments to consider. Two are conditional; one is unconditional. The segment labeled with the condition [d<100] is evaluated first based on the geometry of the two outgoing conditional transition segments. Since the return value of d is 84, the condition [d<100] is true and this transition (to the destination state A.A1) is valid.
- 7 State A.A1 exit actions execute and complete (exitA1()).
- 8 State A.A1 is marked inactive.
- 9 State A.A1 is marked active.
- 10 State A.A1 entry actions execute and complete (entA1()).
- 11 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Example: Transitions from a Common Source to Multiple Destinations

This example shows the semantics of transitions from a common source to multiple destinations.



Initially the Stateflow diagram is asleep. State A is active. Event E_two occurs and awakens the Stateflow diagram. Event E_two is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

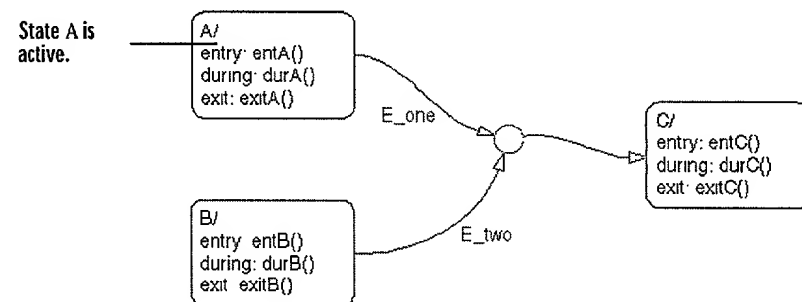
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_two. There is a valid transition segment from state A to the connective junction. Given that the transition segments are equivalently labeled, evaluation begins from a twelve o'clock position on the connective junction and progresses clockwise. The first transition segment labeled with event E_one is not valid. The next transition segment labeled with event E_two is valid. The complete transition from state A to state C is valid.
- 2 State A executes and completes exit actions (exitA()).
- 3 State A is marked inactive.
- 4 State C is marked active.
- 5 State C executes and completes entry actions (entC()).

- 6 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_two.

Example: Transitions from Multiple Sources to a Common Destination

This example shows the semantics of transitions from multiple sources to a single destination.



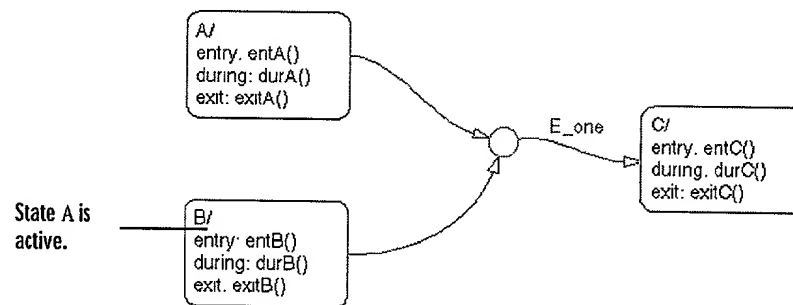
Initially the Stateflow diagram is asleep. State A is active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition segment from state A to the connective junction and from the junction to state C.
- 2 State A executes and completes exit actions (exitA()).
- 3 State A is marked inactive.
- 4 State C is marked active.
- 5 State C executes and completes entry actions (entC()).
- 6 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Example: Transitions from a Source to a Destination Based on a Common Event

This example shows the semantics of transitions from multiple sources to a single destination based on the same event.



Initially the Stateflow diagram is asleep. State B is active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

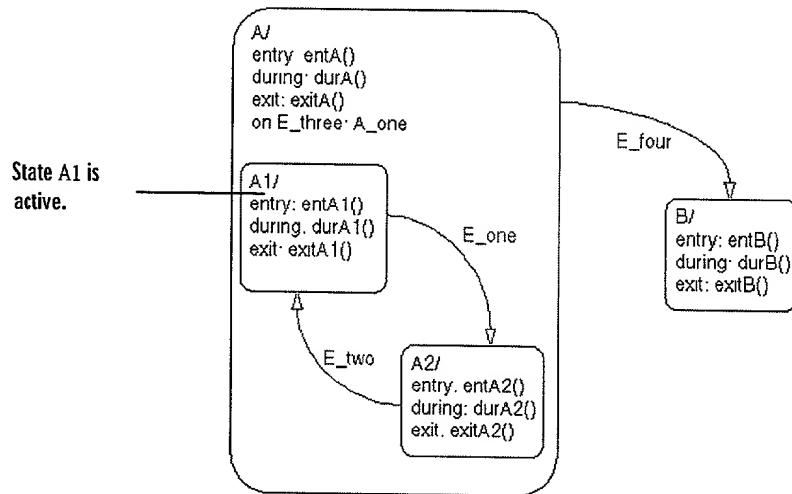
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition segment from state B to the connective junction and from the junction to state C.
- 2 State B executes and completes exit actions (exitB()).
- 3 State B is marked inactive.
- 4 State C is marked active.
- 5 State C executes and completes entry actions (entC()).
- 6 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Event Actions

Example: Event Actions and Superstates

This example shows the semantics of event actions within superstates.



Initially the Stateflow diagram is asleep. State A.A1 is active. Event **E_three** occurs and awakens the Stateflow diagram. Event **E_three** is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of **E_three**. There is no valid transition.
- 2 State A executes and completes during actions (`durA()`).
- 3 State A executes and completes the on event **E_three** action (**A_one**).
- 4 State A checks its children for valid transitions. There are no valid transitions.
- 5 State A1 executes and completes during actions (`durA1()`).
- 6 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

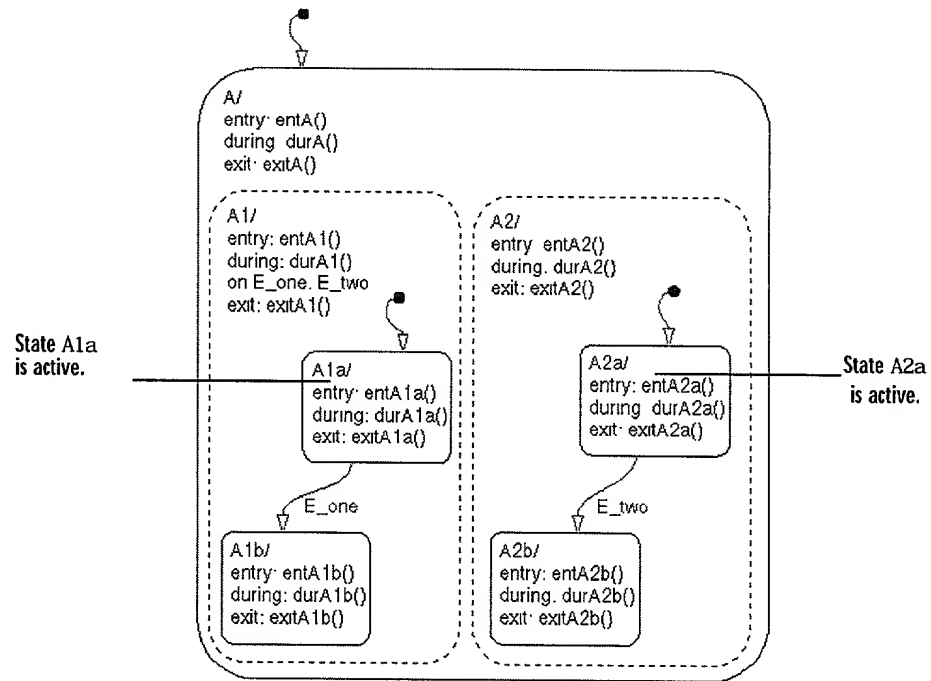
This sequence completes the execution of this Stateflow diagram associated with event E_three.

Figure 8-41: Stateflow diagram associated with event E_three.

Parallel (AND) States

Example: Event Broadcast State Action

This example shows the semantics of event broadcast state actions.

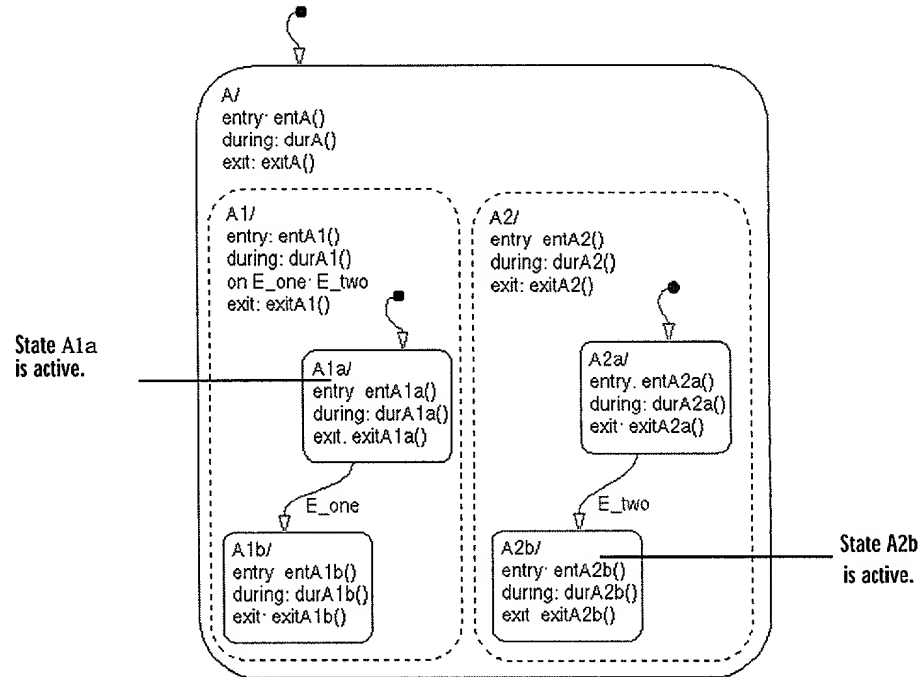


Initially the Stateflow diagram is asleep. Parallel substates A.A1.A1a and A.A2.A2a are active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition at the root level, as a result of `E_one`. There is no valid transition.
- 2 State A executes and completes during actions (`durA()`).
- 3 State A's children are parallel (AND) states. They are evaluated and executed from left to right and top to bottom. State A.A1 is evaluated first.

State A.A1 executes and completes during actions (`durA1()`). State A.A1 executes and completes the `on E_one` action and broadcasts event `E_two`. during and on *event_name* actions are processed based on their order of appearance in the state label.

- a** The broadcast of event `E_two` awakens the Stateflow diagram a second time. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_two`. There is no valid transition.
- b** State A executes and completes during actions (`durA()`).
- c** State A checks its children for valid transitions. There are no valid transitions.
- d** State A's children are evaluated starting with state A.A1. State A.A1 executes and completes during actions (`durA1()`). State A.A1 is evaluated for valid transitions. There are no valid transitions as a result of `E_two` within state A1.
- e** State A.A2 is evaluated. State A.A2 executes and completes during actions (`durA2()`). State A.A2 checks for valid transitions. State A.A2 has a valid transition as a result of `E_two` from state A.A2.A2a to state A.A2.A2b.
- f** State A.A2.A2a exit actions execute and complete (`exitA2a()`).
- g** State A.A2.A2a is marked inactive.
- h** State A.A2.A2b is marked active.
- i** State A.A2.A2b entry actions execute and complete (`entA2b()`). The Stateflow diagram activity now looks like this

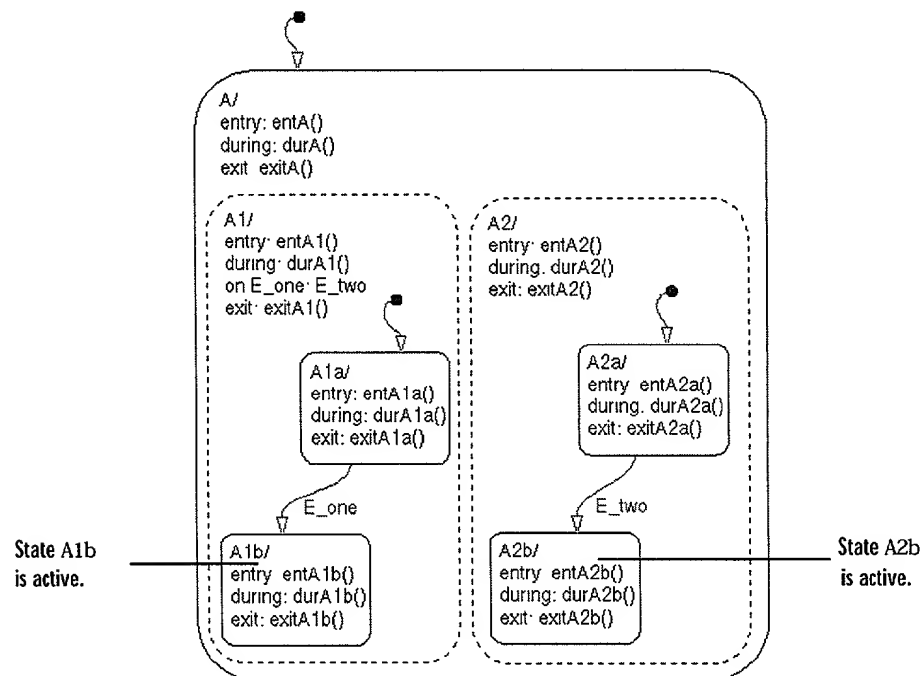


- 4 State A.A1.A1a executes and completes exit actions (exitA1a).
- 5 The processing of E_one continues once the on event broadcast of E_two has been processed. State A.A1 checks for any valid transitions as a result of event E_one. There is a valid transition from state A.A1.A1a to state A.A1.A1b.
- 6 State A.A1.A1a is marked inactive.
- 7 State A.A1.A1b executes and completes entry actions (entA1b()).
- 8 State A.A1.A1b is marked active.
- 9 Parallel state A.A2 is evaluated next. State A.A2 during actions execute and complete (durA2()). There are no valid transitions as a result of E_one.

10 State A.A2.A2b, now active as a result of the processing of the on event broadcast of E_two, executes and completes during actions (durA2b()).

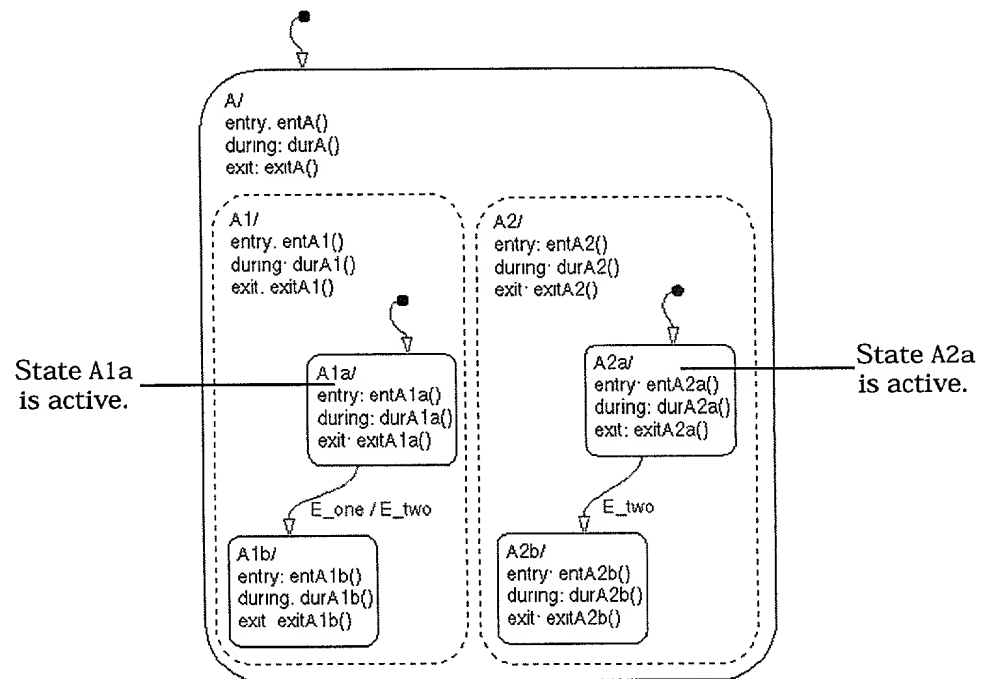
11 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one and the on event broadcast to a parallel state of event E_two. The final Stateflow diagram activity looks like this.



Example: Event Broadcast Transition Action (Nested Event Broadcast)

This example shows the semantics of an event broadcast transition action that includes nested event broadcasts.

**Start of event E_one Processing**

Initially the Stateflow diagram is asleep. Parallel substates A.A1.A1a and A.A2.A2a are active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is no valid transition.
- 2 State A executes and completes during actions (durA()).

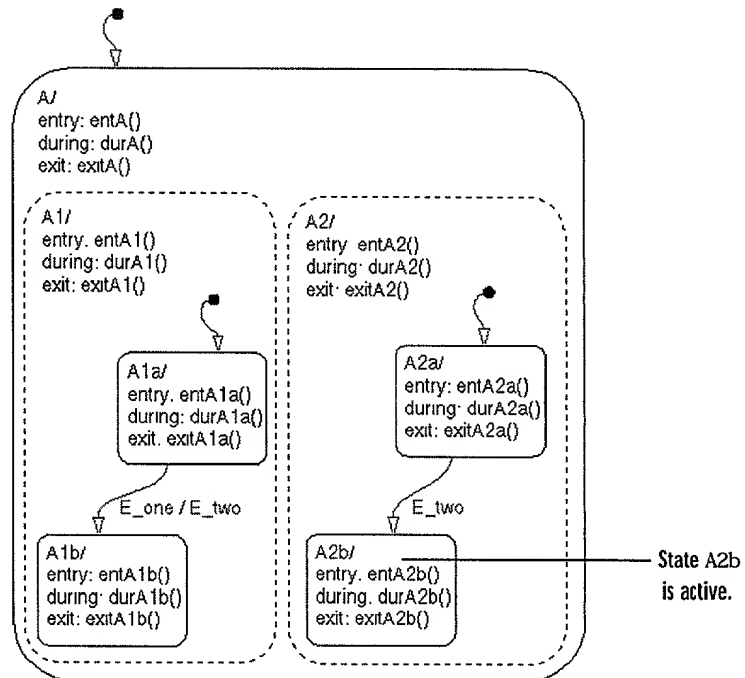
- 3 State A's children are parallel (AND) states. They are evaluated and executed from left to right and top to bottom. State A.A1 is evaluated first. State A.A1 executes and completes during actions (durA1()).
- 4 State A.A1 checks for any valid transitions as a result of event E_one. There is a valid transition from state A.A1.A1a to state A.A1.A1b.
- 5 State A.A1.A1a executes and completes exit actions (exitA1a).
- 6 State A.A1.A1a is marked inactive.

Event E_two Preempts E_one

- 7 Transition action generating event E_two is executed and completed.
 - a The transition from state A1a to state A1b (as a result of event E_one) is now preempted by the broadcast of event E_two.
 - b The broadcast of event E_two awakens the Stateflow diagram a second time. The Stateflow diagram root checks to see if there is a valid transition as a result of E_two. There is no valid transition.
 - c State A executes and completes during actions (durA()).
 - d State A's children are evaluated starting with state A.A1. State A.A1 executes and completes during actions (durA1()). State A.A1 is evaluated for valid transitions. There are no valid transitions as a result of E_two within state A1.
 - e State A.A2 is evaluated. State A.A2 executes and completes during actions (durA2()). State A.A2 checks for valid transitions. State A.A2 has a valid transition as a result of E_two from state A.A2.A2a to state A.A2.A2b.
 - f State A.A2.A2a exit actions execute and complete (exitA2a()).
 - g State A.A2.A2a is marked inactive.
 - h State A.A2.A2b is marked active.
 - i State A.A2.A2b entry actions execute and complete (entA2b()).

Event E_two Processing Ends

The Stateflow diagram activity now looks like this.



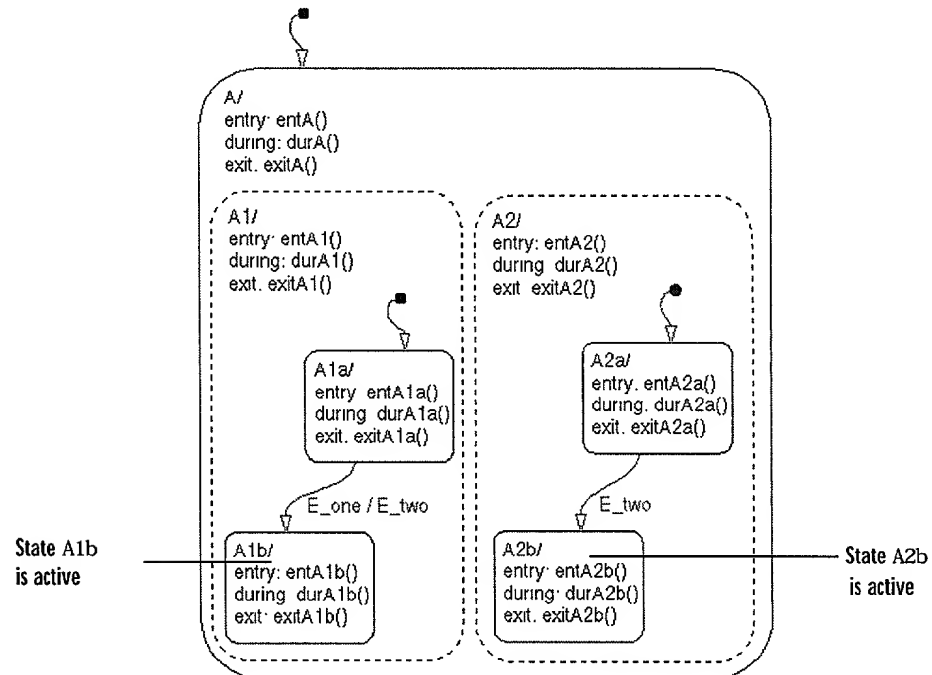
8 State A.A1.A1b is marked active.

Event E_one Processing Resumes

- 9 State A.A1.A1b executes and completes entry actions (entA1b()).
- 10 Parallel state A.A2 is evaluated next. State A.A2 during actions execute and complete (durA2()). There are no valid transitions as a result of E_one.
- 11 State A.A2.A2b, now active as a result of the processing of the transition action event broadcast of E_two, executes and completes during actions (durA2b()).

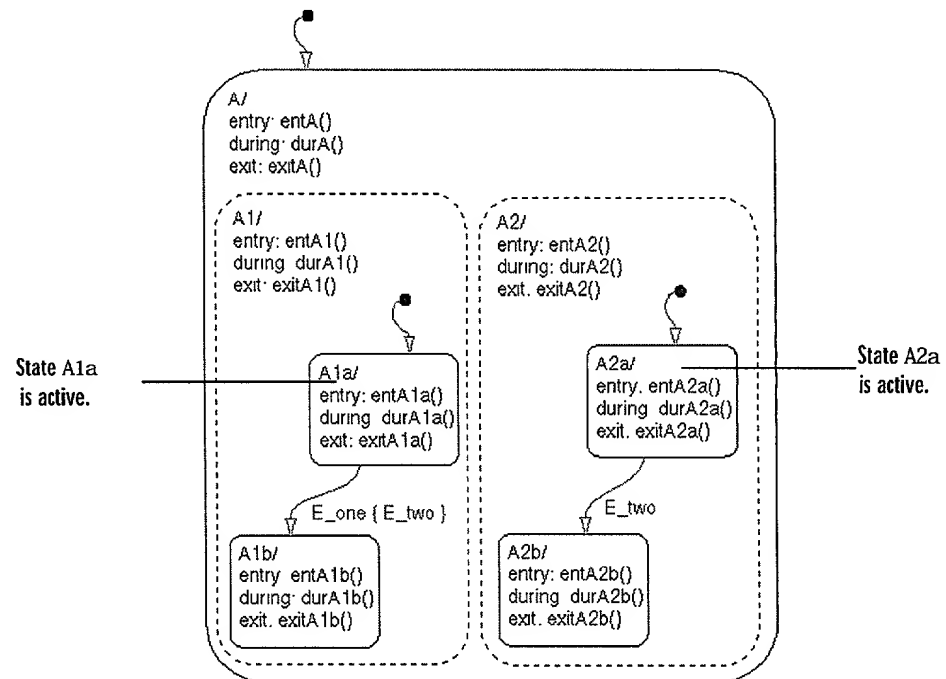
- 12** The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one and the transition action event broadcast to a parallel state of event E_two. The final Stateflow diagram activity now looks like this.



Example: Event Broadcast Condition Action

This example shows the semantics of condition action event broadcast in parallel (AND) states.

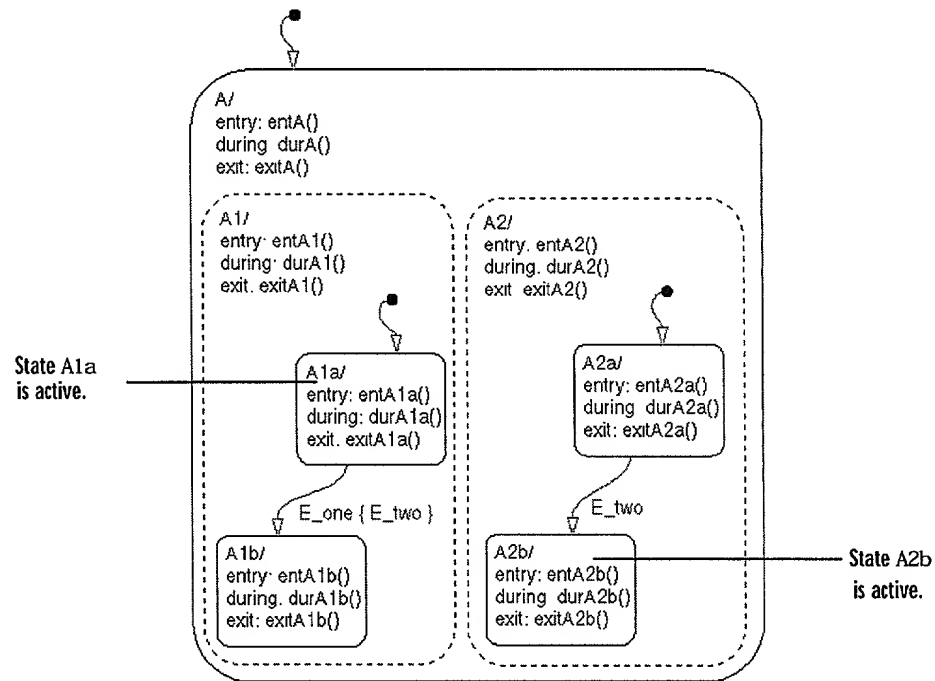


Initially the Stateflow diagram is asleep. Parallel substates A.A1.A1a and A.A2.A2a are active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is no valid transition.
- 2 State A executes and completes during actions (`durA()`).
- 3 State A's children are parallel (AND) states. Parallel states are evaluated and executed from top to bottom. In the case of a tie, they are evaluated from left to right. State A.A1 is evaluated first. State A.A1 executes and completes during actions (`durA1()`).

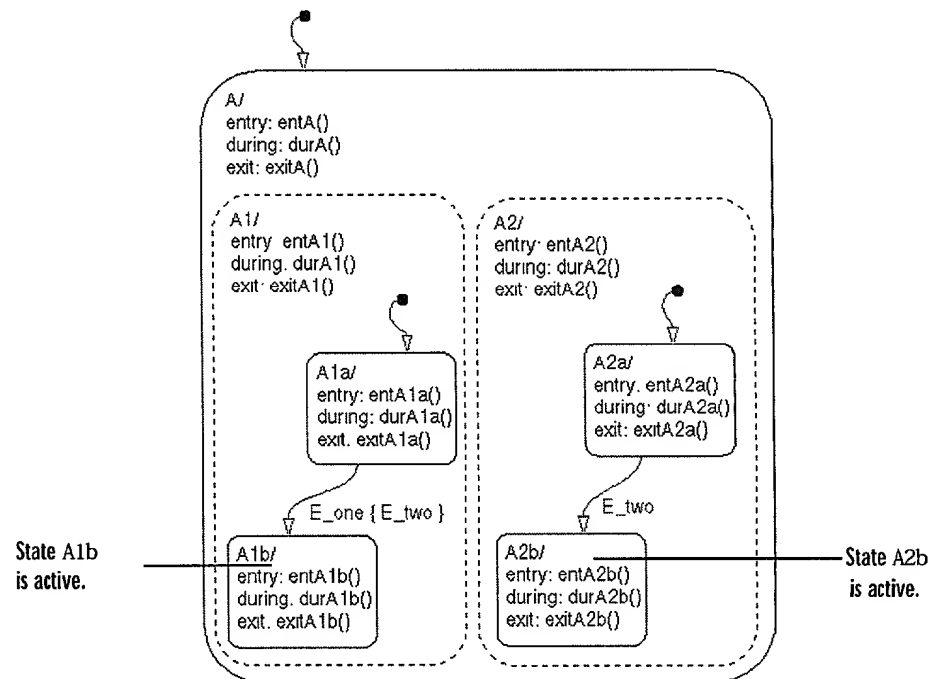
- 4** State A.A1 checks for any valid transitions as a result of event E_one. There is a valid transition from state A.A1.A1a to state A.A1.A1b. There is also a valid condition action. The condition action event broadcast of E_two is executed and completed. State A.A1.A1a is still active.
- a** The broadcast of event E_two awakens the Stateflow diagram a second time. The Stateflow diagram root checks to see if there is a valid transition as a result of E_two. There is no valid transition.
 - b** State A executes and completes during actions (durA()).
 - c** State A's children are evaluated starting with state A.A1. State A.A1 executes and completes during actions (durA1()). State A.A1 is evaluated for valid transitions. There are no valid transitions as a result of E_two within state A1.
 - d** State A.A2 is evaluated. State A.A2 executes and completes during actions (durA2()). State A.A2 checks for valid transitions. State A.A2 has a valid transition as a result of E_two from state A.A2.A2a to state A.A2.A2b.
 - e** State A.A2.A2a exit actions execute and complete (exitA2a()).
 - f** State A.A2.A2a is marked inactive.
 - g** State A.A2.A2b is marked active.
 - h** State A.A2.A2b entry actions execute and complete (entA2b()).

The Stateflow diagram activity now looks like this.



- 5 State A.A1.A1a executes and completes exit actions (exitA1a).
- 6 State A.A1.A1a is marked inactive.
- 7 State A.A1.A1b executes and completes entry actions (entA1b()).
- 8 State A.A1.A1b is marked active.
- 9 Parallel state A.A2 is evaluated next. State A.A2 during actions execute and complete (durA2()). There are no valid transitions as a result of E_one.
- 10 State A.A2.A2b, now active as a result of the processing of the condition action event broadcast of E_two, executes and completes during actions (durA2b()).
- 11 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

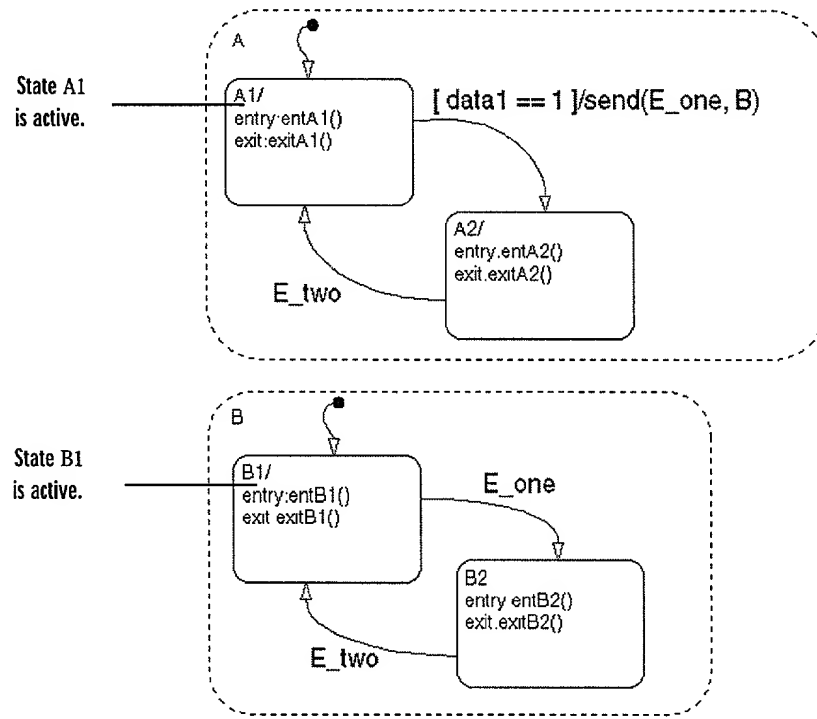
This sequence completes the execution of this Stateflow diagram associated with event E_one and the condition action event broadcast to a parallel state of event E_two. The final Stateflow diagram activity now looks like this.



Directed Event Broadcasting

Example: Directed Event Broadcast Using send

This example shows the semantics of directed event broadcast using `send(event_name, state_name)` in a transition action.



Initially the Stateflow diagram is asleep. Parallel substates A.A1 and B.B1 are active. By definition, this implies parallel (AND) superstates A and B are active. An event occurs and awakens the Stateflow diagram. The condition `[data1==1]` is true. The event is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of the event. There is no valid transition.

- 2 State A checks for any valid transitions as a result of the event. Since the condition `[data1==1]` is true, there is a valid transition from state A.A1 to state A.A2.
- 3 State A.A1 exit actions execute and complete (`exitA1()`).

Start of E_one Event Processing

- 4 State A.A1 is marked inactive.
- 5 The transition action, `send(E_one, B)` is executed and completed.
 - a The broadcast of event `E_one` awakens state B. (This is a nested event broadcast.) Since state B is active, the directed broadcast is received and state B checks to see if there is a valid transition. There is a valid transition from B.B1 to B.B2.
 - b State B.B1 executes and completes exit actions (`exitB1()`).
 - c State B.B1 is marked inactive.
 - d State B.B2 is marked active.
 - e State B.B2 executes and completes entry actions (`entB2()`).

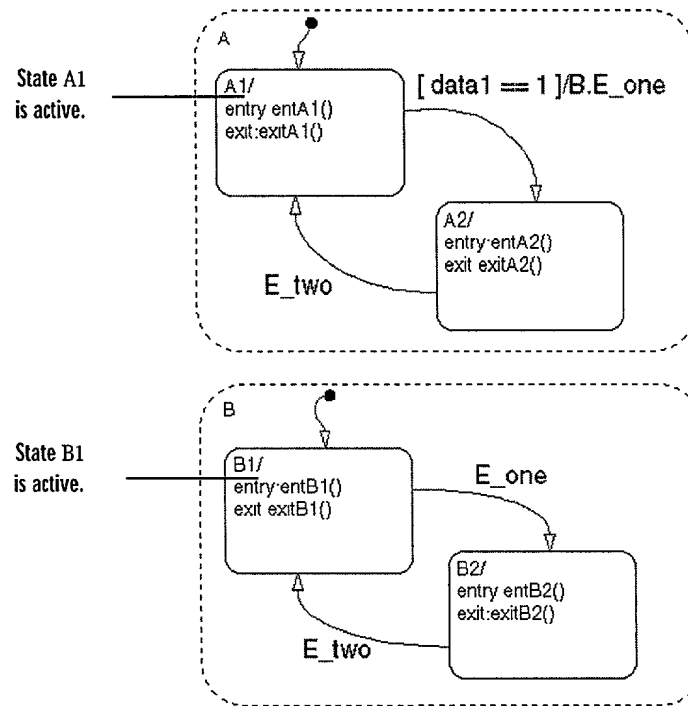
End of Event E_one Processing

- 6 State A.A2 is marked active.
- 7 State A.A2 entry actions execute and complete (`entA2()`).

This sequence completes the execution of this Stateflow diagram associated with an event broadcast and the directed event broadcast to a parallel state of event `E_one`.

Example: Directed Event Broadcasting Using Qualified Event Names

This example shows the semantics of directed event broadcast using a qualified event name in a transition action.



Initially the Stateflow diagram is asleep. Parallel substates A.A1 and B.B1 are active. By definition, this implies parallel (AND) superstates A and B are active. An event occurs and awakens the Stateflow diagram. The condition `[data1==1]` is true. The event is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of the event. There is no valid transition.
- 2 State A checks for any valid transitions as a result of the event. Since the condition `[data1==1]` is true, there is a valid transition from state A.A1 to state A.A2.

- 3** State A.A1 exit actions execute and complete (`exitA1()`).
- 4** State A.A1 is marked inactive.
- 5** The transition action, a qualified event broadcast of event `E_one` to state B (represented by the notation `B.E_one`), is executed and completed.
 - a** The broadcast of event `E_one` awakens state B. (This is a nested event broadcast.) Since state B is active, the directed broadcast is received and state B checks to see if there is a valid transition. There is a valid transition from B.B1 to B.B2.
 - b** State B.B1 executes and completes exit actions (`exitB1()`).
 - c** State B.B1 is marked inactive.
 - d** State B.B2 is marked active.
 - e** State B.B2 executes and completes entry actions (`entB2()`).
- 6** State A.A2 is marked active.
- 7** State A.A2 entry actions execute and complete (`entA2()`).

This sequence completes the execution of this Stateflow diagram associated with an event broadcast using a qualified event name to a parallel state.

Execution Order

Overview

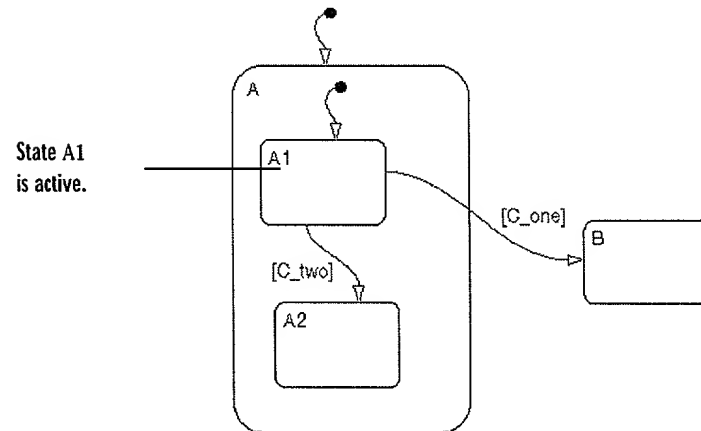
In a single processor environment, sequential execution order is the only option. In this case, it may be necessary for you to know the implicit ordering determined by a Stateflow diagram. The ordering is specific to transitions originating from the same source. Knowing the order of execution for Stateflow diagrams with more than one parallel (AND) state may be important.

Do not design your Stateflow diagram based on an expected execution order.

Execution Order Guidelines

Execution order of transitions originating from the same source is based on these guidelines. The guidelines appear in order of their precedence:

- 1 Transitions are evaluated, based on hierarchy, in a top-down manner. In this example, when an event occurs and state A.A1 is active, the transition from state A.A1 to state B is valid and takes precedence over the transition from state A.A1 to state A.A2 based on the hierarchy.

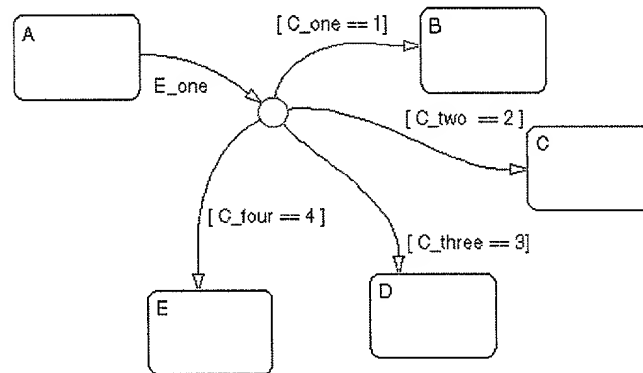


- 2 Transitions are evaluated based on their labels.
 - a Labels with events and conditions
 - b Labels with events

- c Labels with conditions
 - d No label
- 3 Equivalent transitions (based on their labels) are evaluated based on the geometry of the outgoing transitions. The geometry of junctions and states is considered separately.

Junctions

Multiple outgoing transitions from junctions that are of equivalent label priority are evaluated in a clockwise progression starting from a twelve o'clock position on the junction.



In this example, the transitions are of equivalent label priority. The conditions $[C_three == 3]$ and $[C_four == 4]$ are both true. Given that, the outgoing transitions from the junction are evaluated in this order:

- 1 A → B

Since the condition $[C_one == 1]$ is false, this transition is not valid.

- 2 A → C

Since the condition $[C_two == 2]$ is false, this transition is not valid.

3 A → D

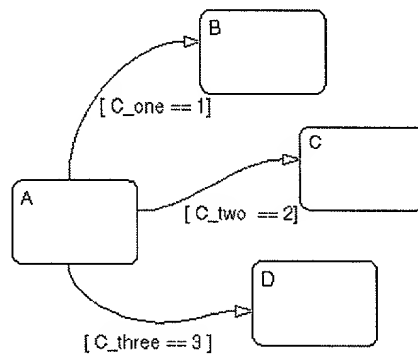
Since the condition $[C_three == 3]$ is true, this transition is valid and is taken.

4 A → E

This transition, even though it too is valid, is not evaluated since the previous transition evaluated was valid.

States

Multiple outgoing transitions from states that are of equivalent label priority are evaluated in a clockwise progression starting at the upper, left corner of the state.



In this example, the transitions are of equivalent label priority. The conditions $[C_two == 2]$ and $[C_three == 3]$ are both true and $[C_one == 1]$ is false. Given that, the outgoing transitions from the state are evaluated in this order:

1 A → B

Since the condition $[C_one == 1]$ is false, this transition is not valid.

2 A → C

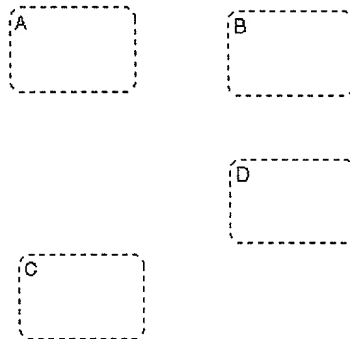
Since the condition $[C_two == 2]$ is true, this transition is valid and is taken.

3 A → D

This transition, even though it too is valid, is not evaluated since the previous transition evaluated was valid.

Parallel (AND) States

Parallel (AND) states are evaluated and executed first from top to bottom and then from left to right in the case of a tie. In this example, assuming that A and B, and C and D are exactly equivalent from top-down, the parallel (AND) states are executed in this order: A, B, D, C.



Semantic Rules Summary

Entering a Chart

The set of default flow paths is executed (see “Executing a Set of Flow Graphs” on page 8-63). If this does not cause a state entry and the chart has parallel decomposition, then each parallel state is entered (see “Entering a State”).

If executing the default flow paths does not cause state entry, a state inconsistency error occurs.

Executing an Active Chart

If the chart has no states, each execution is equivalent to initializing a chart. Otherwise, the active children are executed. Parallel states are executed in the same order that they are entered.

Entering a State

- 1 If the parent of the state is not active, perform steps 1-4 for the parent.
- 2 If this is a parallel state, check that all siblings with a higher (i.e., earlier) entry order are active. If not, perform all entry steps for these states first.
- 3 Mark the state active.
- 4 Perform any entry actions.
- 5 Enter children, if needed:
 - a If the state contains a history junction and there was an active child of this state at some point after the most recent chart initialization, perform the entry actions for that child. Otherwise, execute the default flow paths for the state.
 - b If this state has parallel decomposition, i.e., has children that are parallel states, perform entry steps 1-5 for each state according to its entry order.
- 6 If this is a parallel state, perform all entry actions for the sibling state next in entry order if one exists.

- 7 If the transition path parent is not the same as the parent of the current state, perform entry steps 6 and 7 for the immediate parent of this state.

Executing an Active State

- 1 The set of outer flow graphs is executed (see “Executing a Set of Flow Graphs”). If this causes a state transition, execution stops. (Note that this step is never required for parallel states)
- 2 During actions and valid on-event actions are preformed.
- 3 The set of inner flow graphs is executed. If this does not cause a state transition, the active children are executed, starting at step 1. Parallel states are executed in the same order that they are entered.

Exiting an Active State

- 1 If this is a parallel state, make sure that all sibling states that were entered after this state have already been exited. Otherwise, perform all exiting steps on those sibling states.
- 2 If there are any active children perform the exit steps on these states in the reverse order they were entered.
- 3 Perform any exit actions.
- 4 Mark the state as inactive.

Executing a Set of Flow Graphs

Flow graphs are executed by starting at step 1 below with a set of starting transitions. The starting transitions for inner flow graphs are all transition segments that originate on the respective state and reside entirely within that state. The starting transitions for outer flow graphs are all transition segments that originate on the respective state but reside at least partially outside that state. The starting transitions for default flow graphs are all default transition segments that have starting points with the same parent:

- 1 A set of transition segments is ordered.

- 2 While there are remaining segments to test, a segment is tested for validity. If the segment is invalid, move to the next segment in order. If the segment is valid, execution depends on the destination:

States

- a No more transition segments are tested and a transition path is formed by backing up and including the transition segment from each preceding junction until the respective starting transition.
- b The states that are the immediate children of the parent of the transition path are exited (see "Exiting an Active State").
- c The transition action from the final transition segment is executed.
- d The destination state is entered (see "Entering a State").

Junctions with no outgoing transition segments

Testing stops without any states being exited or entered.

Junctions with outgoing transition segments

Step 1 is repeated with the set of outgoing segments from the junction.

- 3 After testing all outgoing transition segments at a junction, back up the incoming transition segment that brought you to the junction and continue at step 2, starting with the next transition segment after the back up segment. The set of flow graphs is done executing when all starting transitions have been tested.

Executing an Event Broadcast

Output edge trigger event execution is equivalent to changing the value of an output data value. All other events have the following execution:

- 1 If the *receiver* of the event is active, then it is executed (see "Executing an Active Chart" on page 8-62 and "Executing an Active State" on page 8-63). (The event *receiver* is the parent of the event unless the event was explicitly directed to a *receiver* using the `send()` function.)

If the receiver of the event is not active, nothing happens.

- 2 After broadcasting the event, the broadcaster performs early return logic based on the type of action statement that caused the event.

Action Type	Early Return Logic
State Entry	If the state is no longer active at the end of the event broadcast, any remaining steps in entering a state are not performed.
State Exit	If the state is no longer active at the end of the event broadcast, any remaining exit actions and steps in state transitioning are not performed.
State During	If the state is no longer active at the end of the event broadcast, any remaining steps in executing an active state are not performed.
Condition	If the origin state of the inner or outer flow graph or parent state of the default flow graph is no longer active at the end of the event broadcast, the remaining steps in the execution of the set of flow graphs are not performed.
Transition	If the parent of the transition path is not active or if that parent has an active child, the remaining transition actions and state entry are not performed.

